Al-Azhar University
Faculty of Engineering
Department of Electrical Engineering
Electronics and Communication Engineering

Design and Verification of a Custom RISC-V Vector ISA Extension for AI Acceleration

Prepared By:

Abdelrahman Taha Mohamed 2022109
Ahmed Abdelhameed Mahmoud 2022016
Mohamed Emadeldin Ali Sayed 2022120
Omar Ahmad Abd El-Hamid Amin 2022421
Ramadan Khaled Younis 2021120
Yusuf Mohamed Elsayed 2022103

Supervisor: Dr. Abdallah Mohamed Hassan

Sponsorship: STMicroelectronics

A Graduation Project submitted to the Department of Electrical Engineering in partial fulfillment of the requirements for the degree of B.Sc. in Electronics and Communication Engineering

Cairo
July, 2026

Abstract

The fast spread of artificial intelligence has created real pressure to run deep learning inference directly on edge devices, where tight limits on power, area, and cost rule out the hardware that normally does this work. Open, customizable processor architectures have become attractive in this context. The RISC-V instruction set architecture has grown rapidly and now powers around a quarter of new processor silicon, with edge AI cited as one of the main drivers. Unlike the dominant proprietary architectures, x86 in desktops and servers and Arm in mobile and embedded systems, RISC-V is fully open and royalty-free. Designers can extend the instruction set and modify the core without a licensing agreement. The CV32E40P, an open-source 32-bit RISC-V processor from the OpenHW Group, is built for resource-constrained edge devices and provides a practical starting point for this kind of extension.

This project extends the CV32E40P scalar core with a custom RISC-V Vector (RVV) co-processor. Rather than implementing the full RVV specification, we selected only the vector instructions needed for AI inference kernels: widening multiply-accumulate, reductions, element-wise comparison and merge, piecewise-linear function evaluation, and narrowing shifts for requantization. On top of this hardware, we built a kernel library covering convolution, nonlinear activation functions, general matrix multiplication, pooling, and requantization. The kernels were benchmarked on two representative workloads: VWW MobileNetV1, a lightweight CNN for embedded vision, and a single GPT-2 decoder block, a transformer-based sequence-modeling layer.

The project worked bottom-up. We analyzed deep learning workloads to identify the dominant computational patterns, profiled the RVV specification to select the minimum sufficient instruction subset, implemented the co-processor in SystemVerilog, and verified it at three levels: block-level isolation testing, instruction-level RTL verification, and kernel-level three-way comparison against a QEMU golden reference. The verified hardware and kernel library were then benchmarked against the baseline scalar core.

VWW MobileNetV1 ran 4.73× faster on the vector implementation than on the scalar baseline. The GPT-2 decoder block ran 7.00× faster. At the kernel level, speedups range from 2.81× for int32 convolution, where a two-stage widening workaround limits throughput, to over 31× for element-wise ReLU. These numbers come from RTL simulation, not from a functional emulator, and each kernel result was validated against an independent golden reference before it was reported.

Acknowledgment

We would like to thank our supervisor, **Dr. Abdallah Mohamed Hassan**, for his guidance throughout this project. His feedback shaped how we approached the technical work and how we handled the problems that came up along the way.

We are also grateful to **STMicroelectronics** for sponsoring the project, and in particular to **Eng. Ahmed Abd El-Salam** (SoC Verification Engineer) and
Eng. Hassan Mostafa (Digital IC Design Engineer).
Their feedback and industry perspective helped us keep the work grounded in how this is actually done in practice.

Finally, we want to acknowledge the team. A year of working through the same problems together, hardware and software both, is what actually got this finished. It gave all of us a clearer picture of what digital design work looks like beyond coursework, and that made the past year worth it.

Contents

Abstract **1**

List of Abbreviations **12**

1 Introduction **14**
1.1 Motivation . . . 15
1.2 Problem Statement . . . 15
1.3 Project Objectives . . . 16
1.4 Summary of Contributions . . . 16
1.5 Team Contribution Statement . . . 17
1.6 Thesis Outline . . . 17

2 Background and Related Work **18**
2.1 The Edge AI Computing Problem . . . 19
2.1.1 Edge AI versus Cloud AI . . . 19
2.1.2 Hardware constraints at the edge . . . 19
2.2 The RISC-V ISA and Open-Source Hardware . . . 19
2.2.1 What RISC-V is . . . 20
2.2.2 The RISC-V instruction set used in this project . . . 20
2.2.3 RISC-V in the industry . . . 20
2.2.4 RISC-V versus proprietary architectures . . . 21
2.2.5 Open-source toolchain and ecosystem . . . 21
2.3 The CV32E40P Baseline Core . . . 22
2.3.1 Instruction set . . . 22
2.3.2 Microarchitecture . . . 22
2.3.3 Why this core fits the project . . . 24
2.4 Vector Processing . . . 24
2.4.1 Vector register files versus packed-SIMD extensions . . . 24
2.5 The RISC-V Vector Extension (RVV) . . . 25
2.5.1 Vector registers and configuration state . . . 25
2.5.2 Instruction categories . . . 25
2.5.3 What this project implements . . . 26
2.6 Computational Primitives of AI Inference . . . 26
2.6.1 Convolution and GEMM . . . 27
2.6.2 Activation functions . . . 27
2.6.3 Pooling . . . 28
2.6.4 Mapping to the vector instruction subset . . . 28

3 VXT Instruction Set Architecture **29**
3.1 Supported Instruction Subset and Selection Rationale . . . 30
3.2 VLEN / SEW / LMUL Support Matrix; Exclusion of Fractional LMUL . . . 32
3.3 Widening / Narrowing SEW $\leq$ 16 Restriction and Its Effect on Kernel Design . . 33
3.4 Instruction Semantics by Class . . . 33
3.4.1 Configuration Instructions . . . 34
3.4.2 Load and Store Instructions . . . 34
3.4.3 Move and Broadcast Instructions . . . 36
3.4.4 Arithmetic Instructions . . . 37
3.4.5 Shift Instructions . . . 39
3.4.6 Widening and Narrowing Instructions . . . 39
3.4.7 Reduction Instructions . . . 40
3.4.8 Mask, Compare, and Merge Instructions . . . 41
3.5 Known ISA-Level Quirks the Programmer Must Account For . . . 43
3.5.1 The e32,m2 floor(vl/2) Erratum . . . 43
3.6 Divergences from the RVV Specification and Justification for Each . . . 43
3.6.1 D-01: vmax / vmin / vmaxu / vminu Tail-Undisturbed Policy . . . 45
3.6.2 D-02 through D-06: Structural Exclusions . . . 45
3.6.3 D-07: e32,m2 floor(vl/2) Erratum . . . 45
3.6.4 D-08: Widening Source SEW $\leq$ 16 Restriction . . . 45
3.6.5 D-09: Segment Memory Omission . . . 45
3.6.6 D-10 and D-11: Absent Instruction Variants . . . 45
3.6.7 D-12: `vwmacc.vx` Scalar Sign-Extension Erratum . . . 46
Summary . . . 46

4 VXT Microarchitecture and Hardware Implementation **47**
4.1 Introduction . . . 49
4.1.1 Design Goals . . . 49
4.2 System-Level Microarchitecture . . . 49
4.2.1 Top-Level Block Diagram . . . 50
4.2.2 Module Inventory and Responsibilities . . . 50
4.2.3 Data and Control Flow . . . 51
4.2.4 Integration with CV32E40P . . . 52
4.3 Instruction Decoder (`vxt_decoder`) . . . 53
4.3.1 Role and Combinational Nature of the Decoder . . . 53
4.3.2 Block Diagram . . . 54
4.3.3 Port-Level Interfaces . . . 54
4.3.4 `funct3`/`funct6` Decode Logic . . . 56
4.3.5 `vsetvli`/`vsetivli` Configuration Decode . . . 57
4.3.6 Special Cases . . . 57
4.3.7 Design Notes and Known Limitations . . . 58
4.4 CSR Unit (`vxt_csr_rvv`) . . . 59
4.4.1 Role and Update Paths . . . 59
4.4.2 Block Diagram . . . 60
4.4.3 Port-Level Interface . . . 60
4.4.4 Implemented CSRs and Bit-Field Maps . . . 62
4.4.5 VL Computation Logic for `vsetvl`/`vsetivli` . . . 63
4.4.6 Fixed-Point Saturation (`vxsat`) Handling . . . 64
4.4.7 Reset Behaviour and Reset State Summary . . . 65
4.4.8 Timing Considerations . . . 65

4.5 Vector Register File (vxt_regfile) . . . 66
4.5.1 Organisation . . . 66
4.5.2 Read/Write Port Structure . . . 67
4.5.3 Write-Timing Considerations . . . 67
4.6 Vector Controller / FSM (vxt_controller) . . . 68
4.6.1 FSM State Overview . . . 68
4.6.2 State Diagram . . . 70
4.6.3 State-by-State Functional Description . . . 72
4.6.4 Instruction Execution Latency . . . 75
4.6.5 Special Write-Enable Paths and Whole-Register-Move Handling . . . 77
4.6.6 Widening Operations and Reduction Handling . . . 77
4.6.7 VTA/VMA Policy Enforcement at the Controller Level . . . 78
4.7 Vector Load/Store Unit (vxt_vlsu) . . . 79
4.7.1 Overview and OBI-Based Transaction Model . . . 79
4.7.2 Block Diagram . . . 80
4.7.3 Byte-Enable / Active-Element Logic (elem_active_w, Masking, Prestart/Tail Handling) . . . 81
4.7.4 Address Generation and Transaction-Valid Logic . . . 82
4.7.5 Store Path (Pass-Through, Byte-Enable Restriction) . . . 83
4.7.6 Strided Load Logic and Window-Batching Optimisation . . . 83
4.7.7 Unit-Stride Load Merge Logic (4-Way Priority Mux) . . . 84
4.7.8 Sequencer FSM (busy_q, req_cnt_q, pack_cnt_q, rdata_q) and OBI Handshake Timing . . . 84
4.7.9 LMUL=2 Multi-Beat Operation at the VLSU Level . . . 85
4.8 Vector ALU (vxt_alu) . . . 86
4.8.1 Overview . . . 87
4.8.2 Block Diagram . . . 88
4.8.3 Port-Level Interfaces . . . 88
4.8.4 Operand-B Multiplexer (op_b_mux) . . . 90
4.8.5 Main Arithmetic Datapath (vxt_alu Top-Level) . . . 91
4.8.6 Multiplier Sub-Module (vxt_mul) . . . 93
4.8.7 Reduction Tree (vxt_reduction_tree) . . . 93
4.8.8 Design Notes and Known Limitations . . . 94
4.9 RVV Policy Implementation Across the Datapath . . . 94
4.9.1 vstart: Prestart Elements . . . 95
4.9.2 vta: Tail Agnostic . . . 95
4.9.3 vma: Mask Agnostic . . . 96
4.9.4 Mask Register Indexing Across LMUL Beats . . . 96
4.10 Arbitration and Bus Integration . . . 96
4.10.1 Why Arbitration Is Needed . . . 96
4.10.2 Arbitration Scheme . . . 97
4.10.3 Timing Stability Guarantees . . . 97
4.11 Key Design Decisions and Trade-offs . . . 98
4.12 Summary . . . 99

5 Verification and Validation Methodology 100
5.1 Introduction . . . 100
5.2 Verification Objectives . . . 101
5.3 Overall Verification Strategy . . . 101
5.4 Instruction-Level RTL Verification . . . 102

5.5 Kernel and Model Verification Methodology . . . 103
5.6 Software Build Flow for RTL Simulation . . . 104
5.7 Golden Reference Generation Using QEMU . . . 104
5.8 RTL Simulation and Memory Dump Collection . . . 105
5.9 Automated Output Comparison . . . 106
5.10 Debug and Failure Diagnosis . . . 107
5.11 Validation Results Summary . . . 108
5.12 Limitations of the Verification Methodology . . . 110
5.13 Chapter Summary . . . 110

6 AI/ML Primitive Design, Implementation, and Benchmark Results 111
6.1 Nonlinear Activation Functions . . . 113
6.1.1 Rectified Linear Unit (ReLU) . . . 113
6.1.2 Gaussian Error Linear Unit (GELU) . . . 116
6.1.3 Sigmoid . . . 119
6.1.4 Softmax . . . 121
6.2 Requantization . . . 125
6.2.1 Overview . . . 125
6.2.2 Software Implementation . . . 125
6.2.3 Benchmark Setup . . . 125
6.2.4 Benchmark Results . . . 126
6.2.5 Performance Analysis . . . 126
6.2.6 Discussion . . . 129
6.2.7 Key Observations . . . 129
6.3 Matrix Multiplication . . . 129
6.3.1 Overview . . . 129
6.3.2 Software Implementation . . . 130
6.3.3 Benchmark Setup . . . 130
6.3.4 Benchmark Results . . . 130
6.3.5 Performance Analysis . . . 134
6.3.6 Discussion . . . 134
6.3.7 Key Observations . . . 135
6.4 Convolution . . . 135
6.4.1 Overview . . . 135
6.4.2 Software Implementation . . . 135
6.4.3 Benchmark Setup . . . 135
6.4.4 Benchmark Results . . . 135
6.4.5 Performance Analysis . . . 138
6.4.6 Discussion . . . 138
6.4.7 Key Observations . . . 138
6.5 Pooling . . . 138
6.5.1 Overview . . . 138
6.5.2 Software Implementation . . . 139
6.5.3 Benchmark Setup . . . 139
6.5.4 Benchmark Results . . . 140
6.5.5 Performance Analysis . . . 140
6.5.6 Discussion . . . 140
6.5.7 Key Observations . . . 141
6.6 Software Benchmark Summary . . . 141
6.6.1 Overall Speedup Comparison . . . 141

6.6.2 Functions Benefiting Most from Vectorization . . . 141
6.6.3 Computationally Intensive Operations . . . 141
6.6.4 Effectiveness of Software Optimizations . . . 143
6.6.5 Overall Conclusions . . . 143
6.6.6 Key Observations, Summary . . . 143

7 Model Benchmarks **145**
7.1 Introduction . . . 146
7.2 Benchmark Metrics . . . 146
7.3 Visual Wake Words MobileNetV1 Benchmark . . . 146
7.3.1 Model Overview and Usage . . . 146
7.3.2 Reason for Selection . . . 146
7.3.3 Implemented Structure . . . 147
7.3.4 Benchmark Results . . . 147
7.4 GPT-2 Single Decoder Block Benchmark . . . 148
7.4.1 Model Overview and Usage . . . 148
7.4.2 Reason for Selection . . . 148
7.4.3 Implemented Structure . . . 149
7.4.4 Benchmark Results . . . 149
7.5 Overall Benchmark Comparison . . . 150
7.6 Chapter Summary . . . 151

8 Conclusion, Limitations, and Future Work **152**
8.1 Summary of the Work . . . 153
8.2 Key Contributions . . . 153
8.3 Key Results . . . 154
8.4 Limitations . . . 154
8.5 Future Work . . . 155
8.5.1 Architecture . . . 155
8.5.2 ISA . . . 155
8.5.3 Software . . . 156
8.5.4 Evaluation . . . 156
8.6 Conclusion . . . 156
8.7 Chapter Summary . . . 157

List of Figures

2.1 Block Diagram of CV32E40P RISC-V Core 22

4.1 Top-level block diagram of the VXT subsystem. Solid arrows represent data paths; dashed arrows represent control signals. The `obi_mux` arbitrates shared memory access between the scalar core and the VLSU. 50
4.2 Internal combinational block diagram of the `vxt_decoder` module, detailing field extraction and control signal generation. 54
4.3 Internal block diagram of `vxt_csr_rvv`, showing the three write-data sources (standard CSR, `vsetvl` configuration decode, ALU saturation) arbitrated onto the register bank, and the combinational read-data path. 60
4.4 Complete FSM state-transition diagram for `vxt_controller`. Solid arcs are unconditional or always-taken transitions; labelled arcs carry the guard condition. The dashed self-loop on ST_MEM_EXEC represents the wait-for-beat-done stall. The dashed self-loop on ST_IDLE represents the scalar-bus-busy hold-off condition. 70
4.5 Internal microarchitecture of `vxt_vlsu`. Eight functional blocks collaborate under the direction of the internal Sequencer FSM. Data flows from the OBI response bus through the Unit-Stride Load Merge logic or the Strided Logic block into the register file write port. Control flows from the Effective Parameters block through the Byte Enable and Address Generation logic to the OBI request port. Dashed borders indicate combinational blocks; solid borders indicate blocks that contain registers. 80
4.6 Internal combinational block diagram of the `vxt_alu` module, detailing operand selection and result computation logic. 88

5.1 Bottom-up verification pyramid: confidence accumulates from instruction-level RTL verification up to full-system confidence. 102
5.2 Instruction-level RTL verification flow: from manually-driven opcodes on an isolated block, through full integration, to a QuestaSim-checked result. 102
5.3 Scalar and vector implementations of the same algorithm feed into the identical compilation and simulation flow. 103
5.4 Software build flow: from C source to instruction and address data loaded into RTL simulation. 104
5.5 Golden reference generation: a QEMU-friendly build of the kernel is executed under QEMU's ISA-level emulation to produce the golden reference file. 105
5.6 RTL simulation and memory dump collection, using the fixed output address described in this section. 106
5.7 Three-way comparison: scalar hardware output, vector hardware output, and the QEMU golden reference are each compared against one another. 107

5.8 Terminal output of `compare_outputs.py` for a representative kernel verification run. All three pairwise comparisons, scalar vs. golden, vector vs. golden, and scalar vs. vector, pass across 4,096 output elements. 108

6.1 ReLU: Vector vs. Scalar clock cycles across input sizes. 114
6.2 ReLU: Speedup (scalar/vector) as a function of input size. 115
6.3 GELU: Vector vs. Scalar clock cycles across input sizes (Profile A, Attention). . 117
6.4 GELU: Speedup as a function of input size (Profile A, Attention). 118
6.5 Sigmoid: Vector vs. Scalar clock cycles across input sizes (Profile B, Classification). 120
6.6 Sigmoid: Speedup as a function of input size (Profile B, Classification). 120
6.7 Softmax: Vector vs. Scalar clock cycles across all input sizes (Profiles A and B combined). 123
6.8 Softmax: Speedup as a function of input size (combined Profiles A and B). . . . 124
6.9 int32 → int16 Requantization: Vector vs. Scalar clock cycles. 127
6.10 int32 → int16 Requantization: Speedup as a function of input size. 127
6.11 int32 → int8 Requantization: Vector vs. Scalar clock cycles. 127
6.12 int32 → int8 Requantization: Speedup as a function of input size. 128
6.13 int16 → int8 Requantization: Vector vs. Scalar clock cycles. 128
6.14 int16 → int8 Requantization: Speedup as a function of input size. 128
6.15 INT8→INT16 Matrix Multiplication (without bias): Vector vs. Scalar clock cycles. 131
6.16 INT8→INT16 Matrix Multiplication (without bias): Speedup per configuration. 131
6.17 INT8→INT16 Matrix Multiplication (with bias): Vector vs. Scalar clock cycles. 132
6.18 INT8→INT16 Matrix Multiplication (with bias): Speedup per configuration. . . 132
6.19 INT8→INT32 Matrix Multiplication (with bias): Vector vs. Scalar clock cycles. 133
6.20 INT8→INT32 Matrix Multiplication (with bias): Speedup per configuration. . . 133
6.21 Conv2D 3×3: Vector vs. Scalar clock cycles. 136
6.22 Conv2D 5×5: Vector vs. Scalar clock cycles. 137
6.23 Convolution speedup across kernel sizes and output precisions. 137
6.24 Pooling dimension-sweep speedup vs. $C = 16$, $H = 32$, $W = 32$ baseline. The W sweep (row width / OW) shows the strong dependence of speedup on output width alignment with VLMAX. 139

7.1 Visual Wake Words MobileNetV1 architecture used in the implemented benchmark flow. 147
7.2 VWW MobileNetV1 benchmark cycle counts. 148
7.3 GPT-2 single decoder block architecture used in the implemented benchmark flow. 149
7.4 GPT-2 decoder block benchmark cycle counts. 150
7.5 Acceleration comparison for the model benchmarks. 150

List of Tables

2.1 Mapping of AI inference primitives to vector instruction categories in the implemented RVV subset. . . . 28

3.1 VXT supported instruction classes . . . 31
3.2 VLMAX as a function of (SEW, LMUL) for VLEN = 128 . . . 32
3.3 VXT shift instruction summary . . . 39
3.4 VXT divergences from the RVV 1.0 specification . . . 44

4.1 Interface signals between CV32E40P and the VXT coprocessor. . . . 52
4.2 Port-level interfaces of `vxt_decoder`, grouped by destination block. . . . 54
4.3 `funct3` operand-category decode. . . . 57
4.4 Port-level interfaces of `vxt_csr_rvv`. . . . 61
4.5 CSRs implemented by `vxt_csr_rvv`. . . . 62
4.6 Bit-field map of `vtype` (`0xC21`, read-only). . . . 63
4.7 Bit-field map of `vcsr` (`0x00F`, read/write). . . . 63
4.8 Three-case `vl` computation algorithm. . . . 64
4.9 Reset state of all `vxt_csr_rvv` state registers. . . . 65
4.10 Read/write port structure of `vxt_regfile`. . . . 67
4.11 FSM states of `vxt_controller`, their entry points, and their roles. . . . 69
4.12 FSM transition condition legend for `vxt_controller`. Each label corresponds to an arc in Figure 4.4. Suffix convention: `I` = from IDLE, `A` = from ALU_EXEC, `M` = from MEM_EXEC, `D` = from DECODE; `R` = to RETIRE, `A`/`M`/`D` = to that state, `I` = self-loop. Registered signals end in `_r`; combinational inputs end in `_i`. 70
4.13 Execution latency by instruction class. R: OBI round-trip per memory beat. S: VLSU busy duration per strided beat. K: multiply pipeline depth (1 for SEW = 8/16 fast path; 2 for SEW = 32; 3 for `vmulh` SEW = 32). NR: number of registers in a whole-register-move group (1–8). Square brackets indicate a state visited only for LMUL = 2. . . . 76
4.14 Strided window-batching OBI transaction count as a function of stride for SEW=8 (`bytes_per_elem=1`). For SEW=16 or SEW=32 the effective window size is $\lfloor 16/\max(\texttt{stride_i}, \texttt{bytes_per_elem}) \rfloor$, so strides below `bytes_per_elem` disable batching entirely. . . . 83
4.15 Sequencer FSM registers in `vxt_vlsu`. . . . 85
4.16 Port-level interfaces of `vxt_alu`, grouped by source/destination block. . . . 88
4.17 Operation classes implemented in the `vxt_alu` top-level datapath. Multiply-class and reduction operations are handled by the `vxt_mul` and `vxt_reduction_tree` sub-modules respectively (Sections 4.8.6 and 4.8.7). . . . 92

5.1 VXT instruction families exercised during instruction-level RTL verification. . . . 103
5.2 Tools used in the software build flow for RTL simulation. . . . 104
5.3 Memory dump files produced per kernel verification run. . . . 106

5.4 Interpreting the pattern of pairwise comparison outcomes. 108
5.5 Failure symptoms encountered during development and their resolution. 109
5.6 Representative kernel-level verification results. S↔G: scalar vs. golden; V↔G: vector vs. golden; S↔V: scalar vs. vector. 109

6.1 ReLU benchmark results, CV32E40P + VXT (VLEN=128, LMUL=2, SEW=8, int8). 114
6.2 GELU benchmark results, Profile A (Attention), Q3.12 int16, LMUL=2, SEW=16. 117
6.3 Sigmoid benchmark results, Profile B (Classification), Q3.12 int16, LMUL=2, SEW=16. 119
6.4 Softmax benchmark results, Profile A (Attention), Q3.12 int16, LMUL=2, SEW=16. 122
6.5 Softmax benchmark results, Profile B (Classification), Q3.12 int16, LMUL=2, SEW=16. 123
6.6 int32 → int16 requantization benchmark results (LMUL=2, SEW=16 output). . 126
6.7 int32 → int8 requantization benchmark results (LMUL=2, SEW=8 output). . . 126
6.8 int16 → int8 requantization benchmark results (LMUL=2, SEW=8 output). . . 126
6.9 Matrix multiplication, int8→int16, no bias (CV32E40P + VXT). 130
6.10 Matrix multiplication, int8→int16, with bias addition. 131
6.11 Matrix multiplication, int8→int32, with bias addition. 131
6.12 Conv2D 3×3 benchmark results. 136
6.13 Conv2D 5×5 benchmark results. 136
6.14 Conv2D 7×7 benchmark results. 136
6.15 Depthwise Conv2D 3×3 benchmark results. 136
6.16 2×2 max pooling (stride 2) benchmark results, CV32E40P + VXT (VLEN=128, SEW=8, LMUL=2, int8). Average speedup: 8.27×. 140
6.17 Summary of peak and minimum speedups for all benchmarked software functions. 142

7.1 VWW MobileNetV1 benchmark results. 147
7.2 GPT-2 single decoder block benchmark results. 149

8.1 Summary of kernel-level speedups. All cycle counts from QuestaSim RTL simulation. 154
8.2 Model-level benchmark results. 154

List of Abbreviations

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
APU	Auxiliary Processing Unit
ASIC	Application-Specific Integrated Circuit
AVL	Application Vector Length
CAGR	Compound Annual Growth Rate
CISC	Complex Instruction Set Computer
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSR	Control and Status Register
DLP	Data-Level Parallelism
DUT	Device Under Test
EX	Execute (pipeline stage)
FFN	Feed-Forward Network
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GCC	GNU Compiler Collection
GELU	Gaussian Error Linear Unit
GEMM	General Matrix-to-Matrix Multiplication
GPU	Graphics Processing Unit
HDL	Hardware Description Language
ID	Instruction Decode (pipeline stage)
IF	Instruction Fetch (pipeline stage)
INT8	8-bit Integer (quantized datatype)
IoT	Internet of Things
IP	Intellectual Property
ISA	Instruction Set Architecture
KWS	Keyword Spotting
LMUL	Length MULtiplier (vector register grouping factor)
LSU	Load/Store Unit
MAC	Multiply-Accumulate
ML	Machine Learning
OBI	Open Bus Interface

PLA	Piecewise-Linear Approximation
QEMU	Quick EMUlator
QKV	Query, Key, Value (attention mechanism components)
ReLU	Rectified Linear Unit
RISC	Reduced Instruction Set Computer
RISC-V	Reduced Instruction Set Computer — fifth open-standard ISA
RTL	Register Transfer Level
RVV	RISC-V Vector Extension
SEW	Selected Element Width
SIMD	Single Instruction, Multiple Data
SoC	System-on-Chip
SV	SystemVerilog
VALU	Vector Arithmetic Logic Unit
VCSR	Vector Control and Status Register unit
VL	Vector Length (active element count)
VLEN	Vector Register Length (in bits; 128 in this design)
VLMAX	Maximum Vector Length (in elements; VLEN $\times$ LMUL/SEW)
VLSU	Vector Load/Store Unit
VMA	Vector Mask Agnostic (execution policy)
VREG	Vector Register File
VTA	Vector Tail Agnostic (execution policy)
VWW	Visual Wake Words
VXT	Vector eXTension (the custom co-processor developed in this project)
WB	Write-Back (pipeline stage)

Chapter 1

Introduction

Contents

1.1 Motivation **15**
1.2 Problem Statement **15**
1.3 Project Objectives **16**
1.4 Summary of Contributions **16**
1.5 Team Contribution Statement **17**
1.6 Thesis Outline **17**

1.1 Motivation

The rapid growth of artificial intelligence has pushed deep learning inference out of the data center and onto the edge. Applications such as keyword spotting, visual wake words, gesture recognition, and on-device language modeling now demand that inference run locally, close to the sensor, rather than on a remote server. This shift is driven by latency, privacy, connectivity, and energy concerns: edge devices cannot always afford the round-trip delay or power cost of sending data to the cloud, and many applications cannot tolerate the privacy exposure of doing so.

At the same time, the devices expected to run this workload are subject to tight constraints on power, silicon area, and cost. These constraints rule out the GPUs and large accelerators used in data-center inference, and push designers toward small, efficient processor cores that can be customized to the target workload. Open instruction set architectures (ISAs) have become central to this effort, since they allow designers to tailor a core's instruction set to the exact computational patterns of their target workload, without licensing fees or vendor lock-in.

RISC-V has emerged as the fastest-growing open alternative to the proprietary architectures that still dominate computing today: x86 (CISC)-based processors in desktops and servers, and Arm in mobile and embedded systems. RISC-V silicon penetration has grown rapidly over the past several years and is now estimated at roughly a quarter of newly manufactured processor silicon, with edge AI cited as one of the primary drivers of that growth. Its open, modular, and royalty-free nature makes it especially attractive for projects such as this one, where the goal is to extend a core's instruction set with custom, application-specific functionality.

The CV32E40P, developed by the OpenHW Group, is an open-source 32-bit RISC-V core purpose-built for exactly this class of resource-constrained edge device. It implements the RV32IMC base ISA and is widely used in industrial and academic edge-AI projects, which makes it a natural and well-supported starting point for this work.

1.2 Problem Statement

Despite the suitability of cores such as the CV32E40P for edge deployment, their baseline scalar instruction set is not well matched to the computational structure of modern deep learning workloads. Convolutions, matrix multiplications, activation functions, pooling, and quantization all operate on large, regular arrays of data, yet a scalar RV32IMC pipeline processes this data one element at a time, with substantial control-flow and loop-overhead cost on top of the actual arithmetic.

The RISC-V Vector (RVV) extension addresses this mismatch in principle, by allowing a single instruction to operate on many data elements at once. In practice, however, the full RVV specification is large, targeting a broad range of use cases far beyond embedded AI inference, and implementing it in full would impose silicon area, power, and verification costs that are difficult to justify on a low-cost edge core. What a small, cost-sensitive AI-inference core actually needs is narrower than what general-purpose vector hardware offers: a lightweight, FPGA-friendly vector extension, scoped specifically to the operations that AI kernels require, integrated into an open core, and validated in hardware against representative workloads. This gap is the problem the project addresses.

clearpage

1.3 Project Objectives

This project sets out to design, implement, and verify a custom RISC-V Vector extension to the CV32E40P core, scoped specifically to accelerate AI inference, and to demonstrate its effectiveness on real workloads. The concrete objectives are to:

- Analyze representative deep learning workloads (convolutional, mobile, and transformer-based) to identify their dominant computational kernels and bottlenecks.
- Evaluate the baseline CV32E40P (RV32IMC) core to quantify its scalar performance and control-flow overhead on these kernels.
- Profile the RVV specification and select a minimal, sufficient subset of vector instructions needed to implement the target AI kernels efficiently.
- Implement the selected RVV subset in SystemVerilog as an extension to the CV32E40P pipeline, and functionally verify it for correct ISA-level behavior.
- Develop a library of custom vector-accelerated AI kernels: convolution, non-linear activation functions, general matrix multiplication (GEMM), pooling, and quantization; and validate them through hardware-software co-simulation.
- Benchmark the complete hardware-software stack against the baseline scalar core in terms of execution speedup, using Visual Wake Words (MobileNet), and a single GPT-2 decoder layer as representative workloads.

1.4 Summary of Contributions

This project contributes a complete, hardware-verified vector-extended RISC-V platform for edge AI inference. Specifically, it contributes:

- **An application-driven RVV subset:** rather than implementing the full RVV specification, we identify and implement only the vector instructions required by AI inference kernels, keeping the extension lightweight and practical for a low-cost edge core.
- **A verified microarchitecture:** the chosen instruction subset is integrated into the CV32E40P pipeline and verified at the RTL level for correct ISA compliance.
- **A reusable AI kernel library:** a set of vector-accelerated kernels (convolution, non-linear activations, GEMM, pooling, and quantization) that together form the computational core of most modern edge AI models, general enough to support convolutional, mobile, and transformer-style networks alike.
- **A hardware-software co-verification flow:** an infrastructure for validating the custom kernels against the RTL model, ensuring that hardware and software behave consistently.
- **A benchmarking framework:** an evaluation methodology and results comparing the vector-extended core against the baseline scalar core across Visual Wake Words, and a GPT-2 decoder layer, quantifying execution speedup on each workload.

1.5 Team Contribution Statement

This project was carried out by a team of six members. Rather than assigning fixed ownership of individual subsystems, the team worked collaboratively across the full stack: RVV microarchitecture and RTL implementation, hardware verification, AI kernel development, the hardware-software co-simulation infrastructure, and system-level benchmarking. Each member contributed across several of these areas, since the hardware, software, and verification workstreams were tightly coupled throughout the project and rarely progressed independently.

1.6 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 1: Introduction Introduces the background on RISC-V, vector processing, and the motivation for accelerating edge AI workloads. It defines the project objectives and constraints.

Chapter 2: Background and Related Work Reviews the necessary background on the RISC-V ISA, the RISC-V Vector extension, the CV32E40P core, and the computational structure of the target AI workloads.

Chapter 3: VXT Instruction Set Architecture Describes the methodology used to p rofile AI workloads and select the RVV instruction subset implemented in this project.

Chapter 4: VXT Microarchitecture and Hardware Implementation Presents the microarchitectural design and SystemVerilog implementation of the vector extension within the CV32E40P pipeline.

Chapter 5: Verification and Validation Methodology Details the verification strategy used to confirm ISA-level correctness of the implemented hardware, from per-block SystemVerilog testbenches through instruction-level RTL checks to full kernel and model validation.

Chapter 6: AI/ML Primitive Design, Implementation, and Benchmark Results Describes the design and implementation of the custom AI kernel library (convolution, activation functions, GEMM, pooling, and quantization) and the hardware-software co-simulation flow used to validate them.

Chapter 7: Model Benchmarks Presents the benchmarking methodology and results across Visual Wake Words, and the GPT-2 decoder layer, covering execution speedup relative to the baseline scalar core.

Chapter 8: Discussion, Conclusion, and Future Work Concludes the thesis and discusses directions for future work.

Chapter 2

Background and Related Work

Contents

2.1 The Edge AI Computing Problem 19
2.1.1 Edge AI versus Cloud AI 19
2.1.2 Hardware constraints at the edge 19
2.2 The RISC-V ISA and Open-Source Hardware 19
2.2.1 What RISC-V is 20
2.2.2 The RISC-V instruction set used in this project 20
2.2.3 RISC-V in the industry 20
2.2.4 RISC-V versus proprietary architectures 21
2.2.5 Open-source toolchain and ecosystem 21
2.3 The CV32E40P Baseline Core 22
2.3.1 Instruction set 22
2.3.2 Microarchitecture 22
2.3.3 Why this core fits the project 24
2.4 Vector Processing 24
2.4.1 Vector register files versus packed-SIMD extensions 24
2.5 The RISC-V Vector Extension (RVV) 25
2.5.1 Vector registers and configuration state 25
2.5.2 Instruction categories 25
2.5.3 What this project implements 26
2.6 Computational Primitives of AI Inference 26
2.6.1 Convolution and GEMM 27
2.6.2 Activation functions 27
2.6.3 Pooling 28
2.6.4 Mapping to the vector instruction subset 28

2.1 The Edge AI Computing Problem

Modern AI applications increasingly need to run directly on embedded hardware, close to where data is collected, rather than on remote cloud servers. This shift is driven by a set of requirements that cloud deployment simply cannot meet: low latency, offline operation, data privacy, and deterministic response times. Understanding why these requirements push inference onto constrained hardware is the starting point for understanding why accelerating that hardware matters.

2.1.1 Edge AI versus Cloud AI

When an AI model runs in the cloud, every inference request involves a full network round trip: raw data travels out to a remote server, the server processes it, and the result travels back. That round trip introduces latency in the range of 100 ms to over 1000 ms, heavily dependent on network speed and server load. For a wide class of tasks, keyword detection, anomaly detection in sensor streams, visual inspection on a production line, that delay is too long. The decision needs to happen in 1 ms to 50 ms, faster than any network can reliably deliver.

Edge AI solves this by moving the model onto the local device. There is no network round trip, no dependence on connectivity, and no latency floor set by distance to a server. Response times are determined by the hardware itself, not by an external service.

Beyond latency, running inference locally matters for two other reasons. First, **privacy**: raw sensor data, whether audio, video, or biometric, never leaves the device. Second, **offline operation**: the system continues to function when network access is unavailable, which matters for industrial, automotive, and remote deployment environments.

2.1.2 Hardware constraints at the edge

Cloud servers run AI on hardware with no meaningful power or area budget: racks of GPUs drawing hundreds of watts, backed by gigabytes of fast memory. Edge devices operate under the opposite constraints. A microcontroller-class processor typically has a power budget under one watt, a few hundred kilobytes of on-chip SRAM, and a few megabytes of flash. There is no GPU, no large cache hierarchy, and no floating-point accelerator.

These constraints are not independent. Widening the datapath to process more data per clock cycle increases silicon area, which raises die cost. Running wider execution units at higher frequencies increases power. Fitting more on-chip memory improves throughput but again increases area and cost. A design for edge inference has to navigate all three simultaneously, which means the right solution is rarely a scaled-down version of a cloud accelerator. It is something purpose-built for the specific computations the workload actually needs.

That is the problem this project addresses. The target workload, deep learning inference across convolutional, mobile, and transformer models, is dominated by a small set of repetitive arithmetic operations. Rather than building a general-purpose accelerator, the approach taken here extends a small, open-source RISC-V scalar core with a vector instruction set scoped specifically to those operations. The result is a hardware-software stack that delivers real inference acceleration within the power and area envelope of an edge device.

2.2 The RISC-V ISA and Open-Source Hardware

An Instruction Set Architecture defines the contract between hardware and software: what operations a processor can perform, how operands are encoded, and what a compiler or assem-

bler can legally emit. For this project, the choice of ISA is not a neutral technical decision. It determines whether the internal pipeline can be legally modified, whether a custom instruction can be added without negotiating a licence, and whether the full software toolchain, compiler, assembler, simulator, and debugger, is freely available. This section explains why RISC-V is the right answer to all three of those questions, and what that means in practice for the work described in this dissertation.

2.2.1 What RISC-V is

RISC-V is an open-standard Reduced Instruction Set Computer (RISC) architecture originally developed at the University of California, Berkeley, and now governed by RISC-V International, a non-profit consortium. The specification is published under permissive open licences: any individual, company, or institution can implement the ISA, modify a core's internals, or add custom instructions without paying royalties or seeking approval from a licence holder.
The ISA is defined in a mandatory base plus optional extensions. The base integer instruction set, `RV32I` for 32-bit implementations, covers integer arithmetic, branching, memory access, and control-flow. Standard extensions are then layered on top: `M` for integer multiplication and division, `C` for compressed 16-bit instruction encoding, `F` and `D` for single and double-precision floating point, and `V` for the vector processing extension this project implements. An implementation declares exactly which extensions it supports, and software compiled for that profile is guaranteed to run correctly on that hardware.
This modularity is not just an organisational convenience. It means a designer can implement only the arithmetic the target workload actually requires, without inheriting the area and power cost of instructions that will never execute. For an embedded edge core with a tight silicon budget, that discipline matters directly.

2.2.2 The RISC-V instruction set used in this project

The scalar host core used in this work implements **RV32IMC**: the 32-bit base integer instruction set, integer multiply/divide, and compressed instructions. There is no hardware floating-point unit; all arithmetic in this project is integer or fixed-point. The vector extension added by this project implements a constrained subset of the `V` standard, giving the combined design the profile **RV32IMCV**, with the custom vector co-processor attached externally to the scalar pipeline at the instruction- decode stage.

2.2.3 RISC-V in the industry

RISC-V is not purely an academic standard. As of late 2025, the SHD Group reported that RISC-V silicon had reached 25% global market penetration, ahead of the 2030 timeline that analysts had previously forecast. The architecture is projected to ship over 21 billion chips by 2031, and the RISC-V SoC market, valued at approximately $6.1 billion in 2023, is forecast to grow to over $90 billion by 2030 at a compound annual growth rate above 47%. The growth is concentrated in precisely the domains this project targets: embedded AI, IoT, and edge inference, where RISC-V's openness and custom-extension support make it more attractive than proprietary alternatives.
Major semiconductor companies have adopted RISC-V for production hardware. Alibaba, Western Digital, and SiFive have all taped out silicon using the architecture. Google's 2025 Edge TPU revision used RISC-V vector extensions and reported a 34% reduction in MobileNet inference latency compared with an ARM Cortex-M55 baseline, which is directly relevant to the class of workload this project targets.

Citing these figures is not simply about status. It establishes that the infrastructure this project depends on, the open compilers, the verified core RTL, the vector extension specification, is maintained by an active industrial ecosystem, not a research archive. When this project uses the GCC/LLVM toolchain to compile RVV intrinsics, or bases its scalar core on the OpenHW Group's CV32E40P, it is drawing on tooling that major semiconductor firms are also using and actively improving.

2.2.4 RISC-V versus proprietary architectures

The two ISAs that dominate the markets closest to this project's target hardware are Arm, which leads the embedded and mobile space, and x86, which dominates desktop and server computing. Both are proprietary: implementing a licensed Arm or x86 core requires signing a commercial agreement, paying upfront fees, and paying per-unit royalties on every chip shipped. Arm's standard architecture licence has historically cost several million dollars upfront, with ongoing royalties that make sense for a company shipping tens of millions of units but not for a graduation project modifying an existing pipeline.
Beyond the commercial barrier, a proprietary licence restricts what can be done with the core at the RTL level. Adding a new instruction, rewiring the decode stage, or inserting a custom co-processor interface all require explicit permission from the IP owner under most proprietary agreements. That restriction would have made the core modifications at the centre of this project legally impossible under an Arm or x86 licence.
From a purely technical standpoint, RISC-V is competitive with both in the embedded domain. The compressed instruction extension (`C`) achieves code density comparable to Thumb-2 on Arm Cortex-M. The modular extension system avoids the backwards-compatibility overhead that makes x86 decode logic significantly more complex than the execution hardware it feeds. None of this means RISC-V is superior to Arm or x86 in every context; it means that for a resource-constrained, open, extensible embedded platform, the trade-offs favour RISC-V. For this project specifically, the combination of zero licensing cost, legal freedom to modify the RTL, an open-source toolchain, and an already-silicon-proven baseline core made RISC-V the only practical choice.

2.2.5 Open-source toolchain and ecosystem

A hardware ISA is only useful if there is software infrastructure to generate code for it. RISC-V's ecosystem has reached the point where the full toolchain needed for this project is freely available and mature:

- **Compiler:** GCC supports RV32IMC and the RISC-V Vector extension. This project compiles all kernel code using the xPack GNU RISC-V GCC toolchain with the flags `-march=rv32imcv_zve32x_zicsr`, targeting the exact instruction subset the hardware implements.
- **Simulator:** QEMU provides a cycle-accurate enough functional model of the RISC-V scalar pipeline to serve as the golden reference against which kernel outputs are validated in this project, before results are compared against RTL simulation.
- **RTL simulation:** QuestaSim is used for gate-level RTL simulation of the full accelerator system.

All of these are used directly in this project. Their availability, under open licences and with active upstream maintenance, is what made it feasible to develop, verify, and benchmark a custom RISC-V vector extension within the scope of a single graduation project.

2.3 The CV32E40P Baseline Core

The scalar processor onto which this project attaches its vector extension is the **CV32E40P**, an open-source 32-bit RISC-V core from the CORE-V family, developed and maintained by the OpenHW Group. It was chosen as the baseline for three reasons: it is fully open-source with no licensing restrictions on RTL modification, it has been verified to silicon by industrial adopters, and its pipeline is simple enough that attaching a custom co-processor at the decode stage does not require re-architecting the host core's control logic.

2.3.1 Instruction set

The CV32E40P implements the **RV32IMC** instruction set profile: the base 32-bit integer instruction set (I), integer multiplication and division (M), and the 16-bit compressed instruction encoding (C). The top- level wrapper, `CV32E40P top`, includes an optional floating-point unit (FPU) as shown in Figure 2.1, but the FPU is not enabled in the configuration used throughout this project. No instruction in the kernel library or benchmark suite requires floating-point arithmetic; all computation is performed in integer or fixed-point. This is consistent with the quantized, integer-only datapath of the vector extension and with the INT8 inference workloads targeted by the project.

2.3.2 Microarchitecture

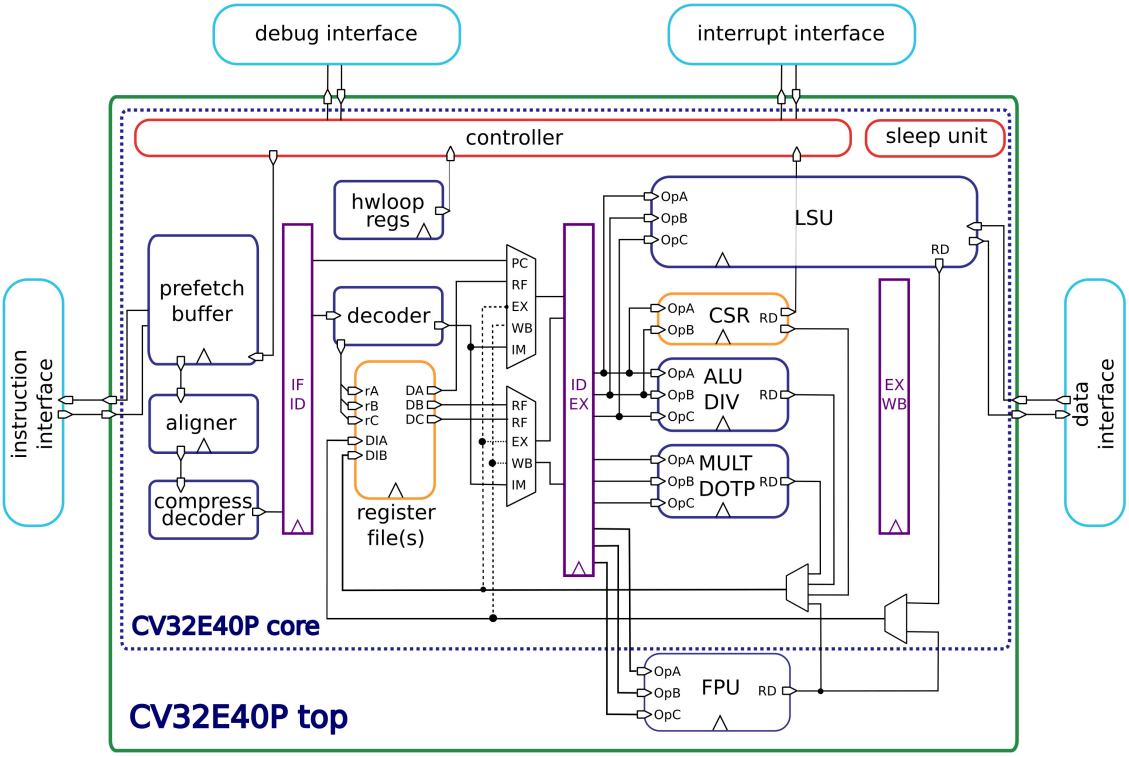

Figure 2.1: Block Diagram of CV32E40P RISC-V Core [1]

Figure 2.1 shows the CV32E40P block diagram. The core implements a four-stage, in-order, single-issue pipeline with the following stages.

Instruction Fetch (IF). The IF stage contains a prefetch buffer that fetches instructions speculatively ahead of the decode stage, an aligner to handle unaligned memory accesses, and a compressed decoder that expands 16-bit compressed (C-extension) instructions into their 32-bit equivalents before they reach the main decoder. A hardware loop register block (`hwloop regs`) is also fed from the ID stage and managed here, supporting the PULP-extended hardware loop instructions present in the core.

Instruction Decode (ID). The ID stage houses the main instruction decoder and the integer register file. The register file exposes three read ports (rA, rB, rC) and three write ports (DA, DB, DC), supporting the three-operand instructions in the PULP extension set. A forwarding network, visible in the diagram as the dotted paths from the ID/EX boundary into the register file write ports, handles data hazards between in-flight instructions and avoids unnecessary stalls for back-to-back dependent operations.

Execute (EX). The EX stage contains three parallel execution units, each taking up to three operand inputs (OpA, OpB, OpC) and producing a result (RD):

- **ALU/DIV:** the main arithmetic-logic unit, handling integer arithmetic, logic, comparison, and division.
- **MULT/DOTP:** the multiplier and dot-product unit, supporting the M-extension multiply instructions and the PULP-extended SIMD dot-product instructions.
- **CSR:** the Control and Status Register unit, managing machine-mode CSRs including the `mstatus`, `mtvec`, and `mepc` registers. In the extended system used in this project, the vector CSRs (`vtype`, `vl`, `vstart`) are managed by the vector extension's own VCSR block rather than this unit.

Load/Store Unit (LSU). The LSU sits alongside the execution units and handles all memory traffic: word, halfword, and byte loads and stores. It connects to the core's data interface (OBI), which in the extended system used in this project is arbitrated by a multiplexer that switches the memory bus between the scalar LSU and the vector extension's VLSU depending on which unit is currently executing.

Write-Back (WB). Results from the execution units and the LSU are written back to the integer register file in the WB stage. The controller, which spans the full width of the core at the top of the diagram, manages pipeline stalls, flushes, hazard detection, and the handshake with external interfaces throughout all four stages.

External interfaces. The core exposes four external interfaces: an instruction interface on the left (OBI, for instruction fetch), a data interface on the right (OBI, for load/store traffic), a debug interface at the top left, and an interrupt interface at the top right. The sleep unit, connected to the controller, manages low-power sleep modes.

2.3.3 Why this core fits the project

The CV32E40P's in-order, single-issue pipeline is the property that makes it practical to extend. In a superscalar or out-of-order core, attaching a co-processor requires interacting with the instruction scheduler, reorder buffer, and register renaming logic, all of which would need modification to recognise and reserve resources for vector instructions. The CV32E40P has none of that complexity. A vector instruction is recognised at the ID stage, the scalar pipeline is stalled via the controller's existing stall mechanism, and control is handed to the vector extension until it signals completion via `instr_done`. The scalar pipeline then resumes as if a multi-cycle scalar instruction had completed. No scheduling logic, no reorder buffer, no renaming: the co-processor attaches cleanly to a single, well-defined point in the pipeline.
The combination of full RTL openness and this pipeline simplicity is what made it feasible to implement, verify, and benchmark a custom vector extension within the scope of a single graduation project.

2.4 Vector Processing

A scalar processor executes one instruction on one pair of operands per clock cycle. When the same arithmetic operation must be applied to a long array of independent data elements, the processor repeats that same instruction once per element, paying the full cost of instruction fetch, decode, and dispatch on every iteration. For a workload such as a convolutional layer that requires hundreds of millions of multiply-accumulate operations, most of the energy and most of the execution time goes to this instruction overhead rather than to the arithmetic itself.
Vector processing addresses this directly. Instead of one operand pair per instruction, a vector instruction names a register that holds a packed array of elements, and the hardware applies the operation to all elements simultaneously. A single instruction fetch and decode drives as many arithmetic operations as there are elements in the vector, which is how instruction overhead gets amortized across useful work.

2.4.1 Vector register files versus packed-SIMD extensions

Two distinct hardware approaches exist for implementing this idea, and they differ in a way that matters for software portability.
Packed-SIMD extensions, such as x86 SSE/AVX or Arm NEON, embed the operand width directly into the instruction encoding. A 128-bit SSE register holds exactly four 32-bit floats or sixteen 8-bit integers, and that is fixed at the ISA level. When the hardware vendor introduces wider registers (256-bit AVX, then 512-bit AVX-512), an entirely new set of instructions must be defined, because the old encodings have the register width baked in. The result is decades of near-duplicate instruction sets that software must explicitly target.
Vector register file architectures take a different approach: the register width, called `VLEN`, is an implementation parameter, not an ISA constant. The ISA specifies how instructions operate on elements within a register, but not how many elements fit in one. Software sets the number of active elements per instruction at run time using a configuration instruction, and hardware processes as many elements as its physical register width allows in one pass. The same binary runs correctly on a narrow implementation processing eight elements per instruction and on a wide one processing sixty-four, without recompilation.
The RISC-V Vector extension (RVV) follows this second approach. This is why the same kernel code written for this project's `VLEN=128` hardware would run correctly on a wider RISC-V implementation: the vector length is configured at run time, and the hardware reports how

many elements it actually processed.

2.5 The RISC-V Vector Extension (RVV)

The RISC-V Vector extension is the official vector instruction set of the RISC-V ISA, ratified as version 1.0 by RISC-V International in late 2021. This is the version targeted throughout this work. The following description of the extension's programming model is drawn from the RVV 1.0 specification.

2.5.1 Vector registers and configuration state

RVV adds thirty-two vector registers, `v0` through `v31`, to the existing scalar register file. The physical width of each register is `VLEN` bits, which is fixed at implementation time. In this project, `VLEN = 128`.
A vector register does not hold a single value. It holds an array of elements, and three parameters govern how that array is interpreted at any given moment.

- **SEW (Selected Element Width)** specifies the width of each element in bits. The RVV specification defines SEW values of 8, 16, 32, and 64. This project supports SEW $\in \{8, 16, 32\}$. At `VLEN=128`, a single vector register therefore holds 16 elements at SEW=8, 8 elements at SEW=16, or 4 elements at SEW=32.

- **LMUL (Length MULtiplier)** controls how many physical vector registers are grouped together to form one logical vector operand. LMUL can be a fraction (to use part of a register) or a whole number (to span multiple registers). Fractional LMUL values are not supported by this project's hardware. LMUL $\in \{1, 2\}$ in this implementation: LMUL=1 uses one physical register per operand, and LMUL=2 groups two consecutive registers into a single 256-bit logical vector, doubling the effective number of elements per instruction. At SEW=8 and LMUL=2 the hardware processes 32 int8 elements in a single instruction.

- **vl (Vector Length)** holds the number of elements the hardware will actually process in the next vector instruction. This is set by the configuration instruction `vsetvli`, which takes the application's requested element count and the desired SEW and LMUL, and writes back the number of elements the hardware can handle given those parameters and its physical `VLEN`. The value returned is at most VLEN $\times$ LMUL/SEW.

  These three parameters, along with two policy bits for tail and mask agnostic behaviour, are encoded in the `vtype` CSR. The current active vector length is held separately in the `vl` CSR. Both are written atomically by `vsetvli` or `vsetivli` and are read implicitly by every vector instruction that executes after them.

2.5.2 Instruction categories

The RVV specification organises its instructions into functional categories. The subset implemented in this project covers the following:

- **Configuration.** `vsetvli` and `vsetivli` set the active `vtype` and `vl` before any vector instruction executes. Every vector code block begins with one of these.

- **Loads and stores.** `vle.v` and `vse.v` transfer a vector register's worth of elements between the register file and consecutive memory locations (unit-stride). `vlse.v` and `vsse.v` extend this to strided access, where consecutive elements are separated by a fixed

byte offset. Strided loads are used in this project to gather rows of convolution windows from the input feature map without reformatting data in memory.

- **Integer arithmetic.** Element-wise add, subtract, and saturating variants thereof, applied across all active elements in a single instruction. Operands may come from a vector register, a scalar register (broadcast), or a small immediate.
- **Integer multiply and widening MAC.** `vmul` and `vmulh` perform element-wise multiplication within the current SEW. `vwmacc.vv` is the central instruction for this project's AI kernels: it multiplies elements of two source registers at SEW and accumulates the products into a destination register at 2×SEW, implementing a widening multiply-accumulate that avoids overflow during dot-product accumulation.
- **Shifts and narrowing.** Logical and arithmetic right shifts, and the narrowing variants `vnsrl` and `vnsra`, which shift and truncate from a wider source to a narrower destination element width. These are used in the requantization kernel to scale int32 accumulator values back to int8 output range.
- **Min/max and comparison.** `vmax`, `vmin` and their unsigned variants operate element-wise. `vmslt` and `vmseq` produce per-element mask results used for conditional selection.
- **Mask and merge.** `vmerge.vvm` selects between two source vectors element-by-element according to a mask in `v0`. This is the instruction used by the non-linear activation kernels to implement piecewise-linear segment selection without branching.
- **Reductions.** `vredsum.vs` and `vredmax.vs` reduce all active elements of a vector register to a single scalar result written into element 0 of the destination. Used by the pooling kernel (max reduction over a window) and the softmax kernel (sum of exponentials).
- **Moves.** `vmv.v.x` broadcasts a scalar register value to all elements of a destination vector. `vmv.s.x` and `vmv.x.s` transfer a single element between the scalar and vector domains.

2.5.3 What this project implements

The full RVV 1.0 specification covers more than the above: indexed (scatter/gather) memory access, whole-register moves, segment loads and stores, and floating-point vector arithmetic, among others. These features are not required by the AI kernels developed in this project, so they are not implemented in the hardware. The implemented subset is exactly the set of instructions that the kernel library uses, no more. This keeps the decoder, vector ALU, and load/store unit lean, matching the practical requirements of a co-processor attached to a constrained edge core.

2.6 Computational Primitives of AI Inference

The execution time of a deep learning inference pass is dominated by a small number of computational patterns. This section describes those patterns and shows how each one maps onto the vector instructions covered in Section 2.5. The mapping is what motivates the choice of instruction subset: every instruction the hardware implements exists because at least one kernel needs it.

2.6.1 Convolution and GEMM

Convolution is the dominant operation in the convolutional layers of MobileNet (VWW), accounting for the large majority of total arithmetic. A standard 2D convolution slides a weight kernel of size $K_H \times K_W$ over an input feature map of $H \times W$ spatial positions and C_{in} input channels, producing one output value per spatial position per output channel.
In this project, convolution is not implemented as a sliding window. The input patches that each kernel position sees are extracted and packed into rows of a matrix using the im2col transformation, and the weight kernels are arranged as columns of a second matrix. The convolution then reduces to a single matrix multiplication, which is also how the fully-connected layers (GEMM) in the feed-forward projections in the GPT-2 decoder layer are computed. Both operations reduce to the same computational pattern: a sequence of dot products between an int8 input vector and an int8 weight vector, with results accumulated into a wider int32 accumulator.
The vector instruction that implements this is `vwmacc.vv`: at SEW=8 and VLEN=128, one instruction multiplies 16 pairs of int8 elements and accumulates their products into 16 int32 accumulators. The inner loop of every convolution and GEMM kernel in this project is a sequence of `vwmacc.vv` instructions, one per tap of the weight kernel or per column of the weight matrix.

2.6.2 Activation functions

Each convolutional or linear layer is followed by a non-linear activation applied element-wise to the output tensor.
ReLU, used after every convolutional layer in MobileNet/VWW, is defined as $f(x) = \max(0, x)$. It evaluates as an element-wise comparison against zero and maps directly onto `vmax.vx` with a zero scalar operand. There is no loop, no lookup table, and no branching: one vector instruction per group of elements.
GELU (Gaussian Error Linear Unit), used in the feed-forward block of the GPT-2 decoder layer, is defined as:

$$\mathrm{GELU}(x) = x \cdot \Phi(x) = 0.5 \cdot x \cdot \left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \tag{2.1}$$

There is no hardware instruction for the Gauss error function on an integer-only core. GELU is instead approximated using a piecewise-linear (PLA) scheme: the function is fit offline with a small number of linear segments over its input domain, and each segment is evaluated at run time using `vmulh` (a high-multiply, equivalent to a fixed-point scaled multiply) followed by a vector add. Segment selection without branching uses the `vmslt`/`vmerge` pattern described in Section 2.5: a mask compare identifies which lanes fall inside each segment boundary, and `vmerge` overwrites those lanes with the segment's result in an outer-to-inner pass so the innermost matching segment wins.
Softmax, used at the output of the GPT-2 attention mechanism, requires computing $e^{x_i} / \sum_j e^{x_j}$ for each element of the input vector. The exponential is approximated with the same PLA scheme used for GELU. The sum of exponentials is computed using `vredsum.vs`, and the division by the sum is avoided entirely: the hardware computes a fixed-point reciprocal of the sum once, then multiplies every element by it using `vmulh`, avoiding the need for a hardware divider.

2.6.3 Pooling

Max pooling downsamples spatial feature maps between convolutional stages. A 2×2 max-pool with stride 2, selects the maximum value within each 2×2 spatial window, reducing the output spatial dimensions by a factor of two in each axis.
The kernel loads each row of a window using strided vector loads and takes the element-wise maximum across rows using `vmax.vv`. The reduction across columns then uses `vredmax.vs` to collapse the remaining elements to the per-window maximum. This is a pure reduction: no multiply, no accumulator widening, no approximation.

2.6.4 Mapping to the vector instruction subset

Table 2.1 summarises how each AI primitive maps to the vector instruction categories implemented in the hardware. The table shows that the four primitive types, dot-product accumulation, element-wise activation, piecewise-linear approximation, and spatial reduction, together require only six instruction categories from the full RVV specification. This is the quantitative justification for implementing a constrained subset rather than the full standard.

Table 2.1: Mapping of AI inference primitives to vector instruction categories in the implemented RVV subset.

Primitive	**Computational pattern**	**Vector instruction(s)**
Convolution / GEMM	Widening dot-product accumulation	`vwmacc.vv`, `vle.v`
ReLU	Element-wise max with zero	`vmax.vx`
GELU / Softmax exp	Piecewise-linear approximation	`vmulh`, `vmslt`, `vmerge`
Softmax normalise	Reciprocal multiply	`vredsum.vs`, `vmulh`
Max pooling	Spatial maximum reduction	`vmax.vv`, `vredmax.vs`
Requantization	Fixed-point shift and narrow	`vnsra`, `vnclip`

Chapter 3

VXT Instruction Set Architecture

Contents

3.1 Supported Instruction Subset and Selection Rationale 30

3.2 VLEN / SEW / LMUL Support Matrix; Exclusion of Fractional LMUL 32

3.3 Widening / Narrowing SEW $\leq$ 16 Restriction and Its Effect on Kernel Design 33

3.4 Instruction Semantics by Class 33

3.4.1 Configuration Instructions 34

3.4.2 Load and Store Instructions 34

3.4.3 Move and Broadcast Instructions 36

3.4.4 Arithmetic Instructions 37

3.4.5 Shift Instructions 39

3.4.6 Widening and Narrowing Instructions 39

3.4.7 Reduction Instructions 40

3.4.8 Mask, Compare, and Merge Instructions 41

3.5 Known ISA-Level Quirks the Programmer Must Account For . . . 43

3.5.1 The e32,m2 floor(vl/2) Erratum 43

3.6 Divergences from the RVV Specification and Justification for Each 43

3.6.1 D-01: vmax / vmin / vmaxu / vminu Tail-Undisturbed Policy 45

3.6.2 D-02 through D-06: Structural Exclusions 45

3.6.3 D-07: e32,m2 floor(vl/2) Erratum 45

3.6.4 D-08: Widening Source SEW $\leq$ 16 Restriction 45

3.6.5 D-09: Segment Memory Omission 45

3.6.6 D-10 and D-11: Absent Instruction Variants 45

3.6.7 D-12: `vwmacc.vx` Scalar Sign-Extension Erratum 46

Summary 46

This chapter defines the software-visible contract of the Vector Extension (VXT) co-processor. Before examining the microarchitecture (which is the subject of Chapter 4) a programmer must have a precise, self-contained reference for every instruction, every legal configuration, every mandatory programming rule, and every divergence from the standard RISC-V Vector (RVV) specification. The sections that follow serve that purpose. This chapter assumes familiarity with RISC-V and with vector processing as covered in Chapter 2. All instruction semantics in this chapter are consistent with the *RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture* [2] (version 1.0, ratified), hereafter referred to as the RVV specification, except where VXT explicitly diverges, as documented in Section 3.6.

3.1 Supported Instruction Subset and Selection Rationale

The full RISC-V Vector specification (RVV 1.0) defines a rich instruction set that covers integer and floating-point arithmetic, cryptographic operations, bit manipulation, segment loads and stores, indexed memory access, and a comprehensive set of mask operations. Implementing the complete standard faithfully on an embedded research platform would require a chip area, verification effort, and design timeline that are incompatible with the graduation project scope. The VXT co-processor therefore targets a targeted subset that provides sufficient coverage to accelerate the classes of neural-network inference kernels (activation functions, convolutions, pooling, and requantisation) that constitute the primary application domain.

Three criteria shaped the selection. Every instruction retained must be exercised by at least one benchmark kernel; dead code in a research implementation is a liability. The subset must be closed, meaning every instruction's operand requirements can be satisfied from within the same subset without invoking excluded instructions. Where the full ISA would demand features, such as floating-point, segment memory operations, or indexed (scatter/gather) addressing, that disproportionately inflate implementation complexity relative to the performance benefit for fixed-point inference, those features are excluded.

Table 3.1 classifies every instruction implemented by VXT. Instructions not present in this table are **not** supported and will be decoded as illegal, potentially causing undefined behaviour if issued.

Several categories were deliberately excluded. Floating-point vector instructions require an FPU datapath and a separate register file format; VXT targets fixed-point (`int8`, `int16`, `int32`) inference exclusively. Segment loads and stores require per-field byte-enable logic and multiple memory transactions per element group; they are absent because none of the benchmark kernels require interleaved multi-field structures. Indexed (scatter/gather) memory operations are omitted because the memory access pattern they produce is irregular and conflicts with the single 128-bit wide OBI data bus design of the memory subsystem. The unsigned saturating add (`vsaddu`) and the signed-unsigned mixed multiply variants (`vwmulsu`, `vwmaccsu`) are also absent from the VXT subset, as the benchmark kernels require only signed fixed-point arithmetic in those instruction categories. Fractional LMUL configurations are excluded; the rationale is discussed in Section 3.2.

Table 3.1: VXT supported instruction classes

Class	**Mnemonic**	**Variants**
Configuration	`vsetvli, vsetivli`	—
Unit-Stride Load	`vle8.v, vle16.v, vle32.v`	—
Unit-Stride Store	`vse8.v, vse16.v, vse32.v`	—
Strided Load	`vlse8.v, vlse16.v, vlse32.v`	—
Strided Store	`vsse8.v, vsse16.v, vsse32.v`	—
Move / Broadcast	`vmv.v.x, vmv.v.i, vmv.s.x, vmv.x.s`	—
Whole-Register Move	`vmv<nr>r.v` (nr = 1, 2, 4, 8)	—
Integer Arithmetic	`vadd, vsub, vrsub, vsadd`	.vv, .vx, .vi
Signed Min/Max	`vmax, vmin`	.vv, .vx
Unsigned Min/Max	`vmaxu, vminu`	.vv, .vx
Multiply	`vmul, vmulh`	.vv, .vx
Shift	`vsrl, vsra, vsll`	.vv, .vx, .vi
Widening Multiply	`vwmul, vwmulu`	.vv, .vx
Widening MACC	`vwmacc`	.vv, .vx
Narrowing Shift	`vnsrl, vnsra`	.wv, .wx, .wi
Narrowing Clip	`vnclip`	.wv, .wx, .wi
Reduction	`vredsum.vs, vredmax.vs`	—
Mask Compare	`vmslt, vmseq`	.vv, .vx; .vi (vmseq only)
Mask Logic	`vmnot.m, vmandn.mm`	—
Merge	`vmerge`	.vvm, .vxm, .vim
Element Index	`vid.v`	—

3.2 VLEN / SEW / LMUL Support Matrix; Exclusion of Fractional LMUL

VXT implements a fixed vector length of **VLEN = 128 bits**. This value is architectural (it cannot be changed at run time) and is reflected in the read-only CSR `vlenb` (vector length in bytes), which reads as 16. The programmer may therefore rely on the fact that a single 128-bit vector register holds exactly 16 bytes, 8 halfwords, or 4 words.

Three values of the Selected Element Width (SEW) are supported: **8 bits (e8)**, **16 bits (e16)**, and **32 bits (e32)**. The 64-bit element width (e64), which the full RVV specification allows when VLEN ≥ 64, is not implemented because the benchmark applications operate exclusively in 32-bit and narrower fixed-point types, and supporting e64 arithmetic would require doubling the multiplier width without a corresponding benefit.

Two values of the register grouping multiplier (LMUL) are supported: **m1** and **m2**. With m1, a vector instruction operates on a single 128-bit register; with m2, the instruction treats two consecutive even-odd register pairs as a single logical 256-bit register group, effectively doubling both the vector length and the number of elements processed per instruction. Per the RVV specification, instructions specifying an LMUL = 2 vector register group with an odd-numbered starting register are reserved. LMUL values of m4 and m8 are not implemented because the eight-element processing width of the ALU datapath already provides the required throughput for all benchmark kernels at m2, and extending LMUL further would require either a wider datapath or a longer multi-beat sequencing path in the controller.

Fractional LMUL (mf2, mf4, mf8) is entirely excluded. In the RVV specification, fractional LMUL reduces the number of elements within a single register and is used to increase the number of available vector register groups when operating on mixed-width values. VXT does not support this feature. From a programming standpoint, if an application needs to process fewer elements than VLMAX, the programmer must rely on `vsetvli` with an explicit application vector length (AVL) that is simply less than VLMAX. The hardware will honour any AVL in the range [0, VLMAX], setting `vl` accordingly, so fractional LMUL is not needed to achieve this effect. Any `vtype` encoding that specifies a fractional LMUL (i.e., `vlmul[2:0]` $\in$ {3'b101, 3'b110, 3'b111}) will cause the `vill` bit to be set, the remaining bits of `vtype` to be zeroed, and `vl` to be set to zero, rendering all subsequent vector instructions that depend on `vtype` illegal until a legal configuration is written.

The complete VLMAX support matrix, showing the maximum number of elements per instruction for each legal (SEW, LMUL) combination, is given in Table 3.2. The formula is VLMAX = LMUL $\times$ VLEN/SEW. Every combination not listed is illegal and will set `vill`.

Table 3.2: VLMAX as a function of (SEW, LMUL) for VLEN = 128

LMUL \ SEW	**e8 (8-bit)**	**e16 (16-bit)**	**e32 (32-bit)**
m1 ($\times$1)	16 elements	8 elements	4 elements
m2 ($\times$2)	32 elements	16 elements	8 elements

The programmer should always use `vsetvli` to obtain the actual `vl` rather than hard-coding VLMAX, particularly when processing arrays whose length is not an exact multiple of VLMAX. The RVV specification guarantees that a value read from the `vl` CSR, when used as the AVL argument to a subsequent `vset{i}vl{i}` instruction with the same `vtype`, will produce the same value of `vl`, provided VLMAX does not change.

3.3 Widening / Narrowing SEW $\leq$ 16 Restriction and Its Effect on Kernel Design

VXT imposes an important restriction on widening multiplication instructions (`vwmul`, `vwmulu`, `vwmacc`): the source element width (the SEW configured when the instruction is issued) must be at most 16 bits. In other words, VXT supports the widening chains **e8** $\rightarrow$ **e16** and **e16** $\rightarrow$ **e32** but does not support the **e32** $\rightarrow$ **e64** chain. The destination operand for widening instructions always occupies elements of width $2 \times \text{SEW}$, and per the RVV specification the effective LMUL of the destination (EMUL) is $2 \times \text{LMUL}$, which means the destination must be a register group of twice the width of the source group.

From the programmer's perspective this restriction is straightforward: widening multiply instructions may only be issued in an epoch where `vsetvli` or `vsetivli` has configured SEW to either 8 or 16. Attempting to issue `vwmul`, `vwmulu`, or `vwmacc` when the active SEW is 32 produces undefined results, as the hardware does not generate an illegal instruction exception for this case.

For narrowing operations (`vnsrl`, `vnsra`, `vnclip`), the source operand is $2 \times \text{SEW}$ wide and the result is SEW wide. The RVV specification defines the shift amount for narrowing operations as using the low $\log_2(2 \times \text{SEW})$ bits of the shift-amount value. All three destination widths are supported: e8 (narrowing from an e16 source), e16 (narrowing from an e32 source), and (as a further case for `vnclip`) e16 from a 32-bit accumulator. The encoding convention in VXT follows RVV exactly: the programmer sets SEW to the *destination* width, and the instruction implicitly reads a source that is twice as wide (denoted by the "`.w`" suffix in the assembly mnemonic).

The practical impact on vector kernel design is as follows. Integer convolution kernels that accumulate into 32-bit accumulators and then quantise back to 8 bits follow a two-stage widening pipeline: first, a sequence of `vwmacc` instructions at $\text{SEW} = 8$ accumulates into e16 partial sums; then, a second epoch at $\text{SEW} = 16$ widens those e16 sums to e32 via a further `vwmacc` sequence. Finally, a `vnclip` instruction at $\text{SEW} = 8$ narrows and clips the e32 accumulator to the e8 output in a single operation, applying the rounding mode from `vxrm` and setting `vxsat` if saturation occurs. This two-stage approach is more verbose than what the full RVV ISA would allow (e32 $\rightarrow$ e64 widening), but it is fully expressible within the VXT subset.

A further corollary for the kernel author concerns register group overlap constraints. Per the RVV specification, a destination vector register group can overlap a source vector register group for a widening instruction only if the overlap is in the highest-numbered part of the destination register group. Concretely, when $\text{LMUL} = \text{m1}$ at $\text{SEW} = 8$, a widening instruction writes an e16 destination at $\text{EMUL} = 2$, occupying two registers. The low-numbered register of that pair cannot overlap `vs1` or `vs2`. The programmer is responsible for allocating distinct, appropriately-aligned register groups for each stage of a widening chain.

3.4 Instruction Semantics by Class

The following subsections describe every instruction in the VXT supported subset. For each instruction, the documentation covers purpose, assembly syntax, operands, element-level behaviour, tail and mask policy interaction, and any VXT-specific notes. Throughout this section, *vd* denotes the destination vector register, *vs1* and *vs2* denote source vector registers, *rs1* denotes a scalar integer register, *imm* or *uimm* denotes an immediate, and *v0* is the dedicated mask register. Tail and mask policy are applied uniformly according to the `vta` and `vma` fields of the active `vtype`, except where noted otherwise.

3.4.1 Configuration Instructions

Configuration instructions are the architectural gateway to all vector execution. They are the only mechanism by which the programmer establishes the active element count (`vl`), element width (`vsew`), and register grouping factor (`vlmul`) before any computational instruction is issued. No vector arithmetic or memory instruction may be executed without a preceding configuration instruction that has established a legal `vtype` and a non-zero `vl`; attempting to do so when `vill` is set raises an illegal-instruction exception.

vsetvli : Set Vector Length (Register AVL)

Syntax: `vsetvli rd, rs1, vtypei`
Operands: *rd*: integer destination register for the new `vl`; *rs1*: integer source register holding the application vector length (AVL); *vtypei*: immediate encoding of `vtype` fields (`ma`, `ta`, `vsew`, `vlmul`).
Behaviour: `vsetvli` atomically updates `vtype` to the requested configuration and computes a new `vl` based on the AVL and the resulting VLMAX. The RVV specification defines three distinct AVL cases controlled by the `rs1` and `rd` register specifiers. When `rs1` $\neq$ x0, the AVL is the unsigned value in `x[rs1]` and `vl` is set according to the standard stripmining constraints (see below). When `rs1` = x0 and `rd` $\neq$ x0, the hardware treats the AVL as the maximum unsigned integer ($\sim$0), and the resulting `vl` is VLMAX for the new configuration. When both `rs1` = x0 and `rd` = x0, the current value of `vl` is used as the AVL and the instruction updates `vtype` without changing `vl`, provided the resulting VLMAX is not reduced; this encoding is reserved if `vill` was set before the instruction. The computed `vl` is written to both the `vl` CSR and the integer register `rd` (when `rd` $\neq$ x0).
Stripmining constraints: The `vl` value satisfies: (1) `vl` = AVL if AVL $\leq$ VLMAX; (2) $\lceil$AVL/2$\rceil \leq$ `vl` $\leq$ VLMAX if VLMAX $<$ AVL $<$ 2$\times$VLMAX; (3) `vl` = VLMAX if AVL $\geq$ 2$\times$VLMAX. VXT always returns `vl` = VLMAX for the second case.
Programmer-visible effect: The CSRs `vl` and `vtype` (including `vill`, `vma`, `vta`, `vsew`, `vlmul`) are updated. `vstart` is reset to zero. If the requested `vtype` is illegal, `vill` is set, the remaining `vtype` bits are zeroed, and `vl` is set to zero.

```
vsetvli t0, a0, e16, m2, ta, ma    # vl = min(a0, 16), SEW=16, LMUL=2
```

vsetivli : Set Vector Length (Immediate AVL)

Syntax: `vsetivli rd, uimm, vtypei`
Operands: *rd*: integer destination register for new `vl`; *uimm*: zero-extended 5-bit unsigned immediate supplying the AVL (range 0–31); *vtypei*: immediate `vtype` encoding.
Behaviour: Functionally identical to `vsetvli` with a non-zero `rs1`, except that the AVL is supplied as a 5-bit zero-extended unsigned immediate. The RVV specification notes that this form provides more compact code when the dimensions of vectors are small and known to fit inside the vector registers. Because the AVL is an unsigned 5-bit field, it can encode values 0–31; for configurations where VLMAX $\leq$ 31, any immediate value up to VLMAX is directly respected. The `rd` = x0, `rs1` = x0 idiom used with `vsetvli` to preserve `vl` is not available with `vsetivli` due to the fixed immediate encoding.

```
vsetivli t0, 8, e8, m1, ta, ma    # vl = min(8, VLMAX_e8m1=16) = 8
```

3.4.2 Load and Store Instructions

VXT supports two memory addressing modes: unit-stride and strided. The element width for memory operations is determined by the *width* field embedded directly in the load/store

instruction encoding, not the active `vtype` SEW. Consequently, a programmer may load e8 elements into a register group while `vtype` is configured at e16, without needing to reconfigure `vtype`. All vector loads and stores may generate and accept a non-zero `vstart` value; the hardware correctly handles resumption at an arbitrary element index.

vle8.v / vle16.v / vle32.v : Unit-Stride Vector Load

Syntax: `vle{8|16|32}.v vd, (rs1), vm`
Operands: *vd*: destination vector register group; *rs1*: base address; *vm*: mask enable bit (`vm` = 0 means masked by `v0`, `vm` = 1 means unmasked).
Behaviour: Loads `vl` elements of the specified width from consecutive memory locations beginning at address `rs1`. Prestart elements ($i <$ vstart) are left undisturbed. Tail elements ($i \geq$ vl) are handled per `vta`. Inactive masked-off elements are handled per `vma`. The VLSU issues a single 128-bit wide OBI transaction per beat, with a byte-enable signal restricting the write to the active element lanes only.

```
vsetvli t0, a1, e8, m1, ta, ma
vle8.v  v0, (a0)              # load t0 bytes from address a0
```

vse8.v / vse16.v / vse32.v : Unit-Stride Vector Store

Syntax: `vse{8|16|32}.v vs3, (rs1), vm`
Operands: *vs3*: source vector register group; *rs1*: base address; *vm*: mask enable bit.
Behaviour: Stores `vl` active elements of the specified width to consecutive memory locations starting at address `rs1`. Only active (non-prestart, non-tail, non-masked) elements are written to memory. Inactive elements do not raise memory exceptions. The VLSU issues a single 128-bit wide OBI write transaction per beat with appropriate byte enables.

vlse8.v / vlse16.v / vlse32.v : Strided Vector Load

Syntax: `vlse{8|16|32}.v vd, (rs1), rs2, vm`
Operands: *vd*: destination register group; *rs1*: base address; *rs2*: stride in bytes (XLEN-wide, treated as a signed offset); *vm*: mask enable bit.
Behaviour: Loads `vl` elements from strided memory locations; element i is loaded from address rs1 + $i \times$ rs2. The VLSU batches multiple elements per OBI transaction when the stride and element size permit packing into a single 128-bit window (e.g., stride = bytes-per-element yields unit packing). For strides that do not permit batching, each element is serviced by a separate OBI transaction. The programmer should be aware that large strides typically result in lower memory throughput than unit-stride operations.

```
li       a2, 4                # stride = 4 bytes
vsetvli  t0, a1, e16, m1, ta, ma
vlse16.v v8, (a0), a2         # load every other int16
```

vsse8.v / vsse16.v / vsse32.v : Strided Vector Store

Syntax: `vsse{8|16|32}.v vs3, (rs1), rs2, vm`
Operands: *vs3*: source register group; *rs1*: base address; *rs2*: stride in bytes; *vm*: mask enable bit.
Behaviour: Stores `vl` active elements to strided memory locations; element i is written to address rs1 + $i \times$ rs2. As with strided loads, the VLSU packs elements into 128-bit OBI write transactions when the stride permits; otherwise each element generates an independent write transaction with appropriate byte enables.

3.4.3 Move and Broadcast Instructions

Move and broadcast instructions transfer data between vector registers, between scalar (integer) registers and vector registers, or fill a vector register with a replicated scalar value. All variants in this group (`vmv.v.x`, `vmv.v.i`, `vmv.s.x`, `vmv.x.s`) are encoded as unmasked instructions (`vm` = 1); the encodings with `vm` = 0 are reserved. The `vmv.v.v` form (copy vector-to-vector) is defined by the RVV specification but is **not** implemented in VXT.

vmv.v.x : Broadcast Scalar to Vector

Syntax: `vmv.v.x vd, rs1`
Behaviour: Copies the low SEW bits of `x[rs1]`, sign-extended to SEW width, into every active element of `vd`. Prestart elements are left undisturbed. Tail elements follow the `vta` policy. The mask register does not affect the destination (unmasked only).

```
vmv.v.x v10, a2      # fill v10[0..vl-1] with sign-extended a2
```

vmv.v.i : Broadcast Immediate to Vector

Syntax: `vmv.v.i vd, imm`
Behaviour: Functionally identical to `vmv.v.x` but the scalar source is a 5-bit sign-extended immediate (range −16 to +15). The decoder expands *imm* to SEW width with sign extension before writing it to every active element of `vd`.

vmv.s.x : Move Scalar to Vector Element 0

Syntax: `vmv.s.x vd, rs1`
Behaviour: Copies the sign-extended low SEW bits of `x[rs1]` into element 0 of `vd`. All elements of `vd` other than element 0 are treated as tail elements and follow the `vta` policy. Per the RVV specification, if `vstart` ≥ `vl` no operation is performed and the destination register is not updated; consequently, when `vl` = 0, no elements are modified. This instruction is principally used to initialise the scalar accumulator for a reduction: the identity element (e.g., 0 for sum, the minimum representable signed value for signed maximum) is placed into element 0 of the destination before issuing a reduction instruction.

```
vmv.s.x v16, zero    # initialise reduction accumulator to 0 in element 0
```

vmv.x.s : Extract Vector Element 0 to Scalar

Syntax: `vmv.x.s rd, vs2`
Behaviour: Sign-extends element 0 of `vs2` to XLEN bits and writes the result into integer register `rd`. Per the RVV specification, this instruction performs its operation unconditionally: it executes and writes `rd` *even if* `vstart` ≥ `vl` *or* `vl` = 0. This is an important distinction from most other vector instructions. No vector state (`vl`, `vtype`) is altered. This instruction is the standard mechanism for reading a reduction result after a `vredsum.vs` or `vredmax.vs` instruction has placed the scalar output in element 0 of the destination register.

```
vredmax.vs v16, v8, v16          # reduction result in v16[0]
vmv.x.s   a3, v16                # extract to scalar register
```

vmv<nr>r.v , Whole Vector Register Move (nr = 1, 2, 4, 8)

Syntax: `vmv1r.v vd, vs2` `vmv2r.v vd, vs2` `vmv4r.v vd, vs2` `vmv8r.v vd, vs2`

Behaviour: Copies NREG whole vector registers from `vs2` to `vd`. The RVV specification defines that these instructions operate as if EEW = SEW, EMUL = NREG, and an effective length of evl = NREG × VLEN/SEW elements. The instructions ignore the current `vl` setting; specifically, the RVV-specified property that no elements are written when `vstart` ≥ `vl` does *not* apply. Instead, no elements are written when `vstart` ≥ evl. Masking is not available; these instructions are always unmasked. The source and destination register groups must be aligned to the NREG boundary (e.g., `vmv2r.v` requires even-numbered starting registers). If `vd` equals `vs2`, the instruction is an architectural NOP but serves as a hint to the implementation to internally re-format the register data for SEW-compatible access.

Typical usage: Copying an accumulator register group between register banks during a multi-stage kernel computation without disturbing `vtype` or `vl`.

3.4.4 Arithmetic Instructions

VXT provides a set of integer arithmetic instructions covering addition, subtraction, reverse subtraction, saturating addition, and element-wise minimum and maximum. Unless noted otherwise, all arithmetic instructions are available in the following operand flavours: `.vv` (vector–vector), `.vx` (vector–scalar register), and `.vi` (vector–sign-extended 5-bit immediate). All instructions honour the active `vl`, `vstart`, `vta`, and `vma` policies. Integer operations wrap around on overflow unless otherwise specified.

vadd : Integer Add

Syntax:
```
vadd.vv vd, vs2, vs1, vm
vadd.vx vd, vs2, rs1, vm
vadd.vi vd, vs2, imm, vm
```

Behaviour: Performs element-wise addition of `vs2` and the second source operand. Results wrap modulo 2^{SEW}. The `.vi` form uses a sign-extended 5-bit immediate as the scalar source.

```
vadd.vv v4, v8, v12      # v4[i] = v8[i] + v12[i]
vadd.vx v4, v8, a0       # v4[i] = v8[i] + a0
```

vsadd : Signed Saturating Add

Syntax:
```
vsadd.vv vd, vs2, vs1, vm
vsadd.vx vd, vs2, rs1, vm
vsadd.vi vd, vs2, imm, vm
```

Behaviour: Performs signed saturating addition. If the true signed sum exceeds the maximum representable value for the current SEW ($2^{SEW-1} - 1$), the result is clamped to that maximum; if the sum is below the minimum (-2^{SEW-1}), the result is clamped to that minimum. When saturation occurs, the `vxsat` bit in the `vcsr` CSR is set. VXT implements the signed variant (`vsadd`) only; the unsigned saturating add (`vsaddu`) is not supported.

```
vsadd.vx v2, v4, a1    # sat-add: v2[i] = sat_signed_SEW(v4[i] + a1)
```

vsub : Integer Subtract

Syntax: `vsub.vv vd, vs2, vs1, vm  vsub.vx vd, vs2, rs1, vm`
Behaviour: Subtracts `vs1` (or `rs1`) from `vs2`, element-wise, with wrapping: vd[i] = vs2[i] − vs1[i].

vrsub : Integer Reverse Subtract

Syntax: `vrsub.vx vd, vs2, rs1, vm  vrsub.vi vd, vs2, imm, vm`
Behaviour: Computes vd[i] = rs1 − vs2[i] (or imm − vs2[i]). The scalar/immediate operand is the minuend. The RVV specification notes that negation can be expressed as an assembler pseudoinstruction `vneg.v vd,vs = vrsub.vx vd,vs,`
`x0`. The instruction is useful for implementing pointwise negation plus offset, as required in the symmetry step of sigmoid and tanh approximations.

```
vrsub.vx v6, v4, a3     # v6[i] = a3 − v4[i]
```

vmax / vmin : Element-Wise Signed Maximum / Minimum

Syntax: `vmax.vv vd, vs2, vs1, vm`
`vmax.vx vd, vs2, rs1, vm`
`vmin.vv vd, vs2, vs1, vm`
`vmin.vx vd, vs2, rs1, vm`

Behaviour: Computes the element-wise signed maximum (`vmax`) or signed minimum (`vmin`). Per the RVV specification, the tail agnostic/undisturbed policy for min/max instructions is governed by the `vta` field. VXT, however, always applies the tail-undisturbed policy for these instructions regardless of the `vta` setting; this is a documented divergence from the RVV specification listed in Section 3.6.
Typical usage: ReLU activation: `vmax.vx vd, vs2, zero` clips all negative elements to zero in a single instruction.

```
vmax.vx v8, v8, zero    # ReLU: v8[i] = max(v8[i], 0)
```

vmaxu / vminu : Element-Wise Unsigned Maximum / Minimum

Syntax: `vmaxu.vv vd, vs2, vs1, vm  vmaxu.vx vd, vs2, rs1, vm`
`vminu.vv vd, vs2, vs1, vm  vminu.vx vd, vs2, rs1, vm`
Behaviour: Computes the element-wise unsigned maximum (`vmaxu`) or unsigned minimum (`vminu`). Elements are compared as unsigned integers; this is the key distinction from the signed `vmax` and `vmin` instructions. As with the signed variants, VXT applies tail-undisturbed policy regardless of `vta`.
Typical usage: Clamping unsigned activation outputs to a maximum representable value, or implementing ReLU on data encoded in an unsigned fixed-point format.

vmul / vmulh : Integer Multiply (Low / High Half)

Syntax: `vmul.vv vd, vs2, vs1, vm  vmul.vx vd, vs2, rs1, vm`
`vmulh.vv vd, vs2, vs1, vm  vmulh.vx vd, vs2, rs1, vm`
Behaviour: The RVV specification defines that single-width multiply instructions perform a SEW×SEW-bit multiply to generate a 2×SEW-bit product, then return one half. `vmul` returns the low SEW bits; `vmulh` returns the high SEW bits of a signed×signed product. The high-half multiply is the key primitive for fixed-point scaling: given two Q-format numbers, their signed high-half product yields the correctly shifted result without an explicit shift instruction, provided the programmer accounts for the implicit division by 2^{SEW}.

```
vmulh.vx v12, v8, a4    # v12[i] = high_SEW(v8[i] * a4)  (Q−format scale)
```

3.4.5 Shift Instructions

VXT implements three shift instructions, each available in `.vv`, `.vx`, and `.vi` operand forms. Per the RVV specification, only the low $\log_2(\text{SEW})$ bits of the shift-amount value are used to control the shift amount. For the immediate forms (`.vi`), the shift amount is encoded as a *zero-extended* (unsigned) 5-bit immediate, in contrast to the sign-extended 5-bit immediates used by arithmetic instructions. Tail and mask policy are applied normally.

Table 3.3: VXT shift instruction summary

Instruction	Operation	Shift Amount Used
`vsrl`	Logical right shift (zero-fill)	low $\log_2(\text{SEW})$ bits of source
`vsra`	Arithmetic right shift (sign-fill)	low $\log_2(\text{SEW})$ bits of source
`vsll`	Logical left shift (zero-fill)	low $\log_2(\text{SEW})$ bits of source

All three shift instructions are frequently used in fixed-point scaling pipelines. For example, after a `vmulh` sequence that multiplies two Q12 numbers, a `vsra` by 2 may be needed to compensate for the fixed-point format shift. The shift amount may also be supplied as a vector operand (via `.vv`), allowing per-element variable shift amounts when quantisation scale factors differ across channels.

```
vsra.vi v8, v8, 2      # arithmetic right shift by 2 (unsigned immediate)
vsrl.vv v4, v4, v6     # logical right shift by per-element amount in v6
```

3.4.6 Widening and Narrowing Instructions

Widening instructions produce a result whose element width is $2 \times \text{SEW}$. Narrowing instructions consume a source with element width $2 \times \text{SEW}$ and produce a result at SEW. Per the RVV specification, the SEW field in the active `vtype` always refers to the *narrow-domain* element width: the destination width for widening instructions and the destination width for narrowing instructions. The restriction that widening is only supported when the narrow-domain $\text{SEW} \leq 16$ is covered in Section 3.3 and applies to all widening instructions in this subsection.

vwmul / vwmulu : Widening Signed / Unsigned Multiply

Syntax: `vwmul.vv vd, vs2, vs1, vm` `vwmul.vx vd, vs2, rs1, vm`
`vwmulu.vv vd, vs2, vs1, vm` `vwmulu.vx vd, vs2, rs1, vm`
Behaviour: `vwmul` computes the full $2 \times \text{SEW}$ signed-signed product of each SEW-wide element pair. `vwmulu` computes the full unsigned-unsigned product. The destination register group `vd` has $\text{EMUL} = 2 \times \text{LMUL}$ and $\text{EEW} = 2 \times \text{SEW}$. Per the RVV overlap constraints, the destination must not overlap `vs1` or `vs2` except in the highest-numbered part of the destination group. For the current VXT LMUL support: at $\text{SEW} = 8$, m1, the destination is an e16, m2 group (two consecutive registers); at $\text{SEW} = 16$, m1, the destination is an e32, m2 group.

vwmacc : Widening Signed Multiply-Accumulate

Syntax: `vwmacc.vv vd, vs1, vs2, vm` `vwmacc.vx vd, rs1, vs2, vm`
Behaviour: Computes $\text{vd}[i] \mathrel{+}= \text{sign_extend}(\text{vs1}[i]) \times \text{sign_extend}(\text{vs2}[i])$ in $2 \times \text{SEW}$ arithmetic. The accumulator `vd` is in the wide domain. Note the operand order: in the `.vv` encoding, `vs1` is the first multiplicand and `vs2` is the second; in the `.vx` encoding, `rs1` provides the scalar first multiplicand and `vs2` provides the vector second multiplicand. The destination `vd` must not overlap `vs1` or `vs2` in the `.vv` case. This instruction is the workhorse of convolution and dense matrix kernels in VXT.

```
# Widening MAC chain: e8,m1 sources -> e16,m2 accumulator in v16
vsetvli t0, vl, e8, m1, ta, ma
vle8.v   v8,  (src_ptr)            # load input slice
vmv.v.x  v10, weight               # broadcast weight
vwmacc.vv v16, v10, v8             # v16 += v10 * v8  (e16 accumulator)
# ... additional taps ...
vmv.v.x  v10, zero                 # VXT_FENCE step 1
vwmacc.vv v16, v10, v8             # VXT_FENCE step 2 (no-op mathematically)
```

vnsrl / vnsra : Narrowing Logical / Arithmetic Right Shift

Syntax: `vnsrl.wv vd, vs2, vs1, vm` `vnsrl.wx vd, vs2, rs1,` `vnsrl.wi vd, vs2, uimm, vm` `vnsra.wv vd, vs2, vs1, vm` `vnsra.wx vd, vs2, rs1, vm` `vnsra.wi vd, vs2, uimm, vm`

Behaviour: The source `vs2` has EEW = 2× SEW (the "`.w`" source suffix denotes this). Per the RVV specification, the low $\log_2(2 \times \text{SEW})$ bits of the shift-amount value are used. Each $2 \times$ SEW element is right-shifted by that amount, and the low SEW bits of the result are written to `vd`. `vnsrl` performs a logical (zero-filling) shift; `vnsra` performs an arithmetic (sign-preserving) shift. These instructions are used in fixed-point requantisation pipelines to scale down accumulated wide sums to narrower output widths. The immediate form uses a zero-extended 5-bit unsigned value as the shift amount.

```
vsetvli t0, vl, e16, m1, ta, ma
vnsra.wi v4, v16, 12    # v4[i] = v16[i][31:0] >> 12 (signed, narrow to 16-bit)
```

vnclip : Narrowing Fixed-Point Clip (Signed)

Syntax: `vnclip.wv vd, vs2, vs1, vm` `vnclip.wx vd, vs2, rs1, vm` `vnclip.wi vd, vs2, uimm, vm`

Behaviour: Performs a right shift of the 2×SEW-wide source element (using the low $\log_2(2\times$ SEW) bits of the shift amount), applies the rounding mode from `vxrm` prior to saturation, and then saturates the rounded result to the signed SEW-wide range before writing to `vd`. If saturation occurs, `vxsat` is set. This instruction combines shifting, rounding, and clipping into a single operation, which is the critical final step in many quantisation pipelines. VXT implements the signed saturating form (`vnclip`) only; the unsigned saturating form (`vnclipu`) is not supported. The rounding mode is always taken from the `vxrm` field of `vcsr`.

```
vsetvli t0, vl, e8, m1, ta, ma
vnclip.wi v0, v16, 8        # shift right 8, round, clip to [-128,127]
```

3.4.7 Reduction Instructions

VXT implements two reduction instructions that collapse an entire vector register group into a scalar result. Per the RVV specification, vector reduction operations take a vector register group of elements and a scalar operand held in element 0 of a *single* vector register (not a vector register group), and produce a scalar result in element 0 of a destination vector register. This means any vector register (regardless of the current LMUL setting) may serve as the scalar source (`vs1`) or scalar destination (`vd`) of a reduction. The remaining elements of `vd` (elements 1 through VLEN/SEW − 1) are treated as tail elements and managed with the current `vta` policy.

Inactive masked-off elements of the source vector register group (`vs2`) are *excluded* from the reduction. The scalar operand in element 0 of `vs1`, however, is always included in the computation regardless of the mask. If `vl` = 0, no operation is performed and the destination register is not updated.
Traps on reduction instructions are always reported with `vstart` = 0. Reductions raise an illegal-instruction exception if `vstart` is non-zero before the instruction begins.

vredmax.vs : Vector Signed Maximum Reduction

Syntax: `vredmax.vs vd, vs2, vs1, vm`
Behaviour: Computes the signed maximum over all active elements of `vs2`, initialised with element 0 of `vs1`. The result, max(vs1[0], vs2[0], vs2[1], ..., vs2[vl − 1]) over active elements, is placed in element 0 of `vd`. All other elements of `vd` are tail. This instruction is used in softmax computation to find the global maximum of the input vector before the exponential normalisation pass.

```
li        t1, -32768              # smallest int16 (for SEW=16)
vmv.s.x   v16, t1                 # seed element 0 of scalar operand
vredmax.vs v16, v8, v16           # v16[0] = max(v16[0], v8[*])
vmv.x.s   a3, v16                 # extract result to scalar
```

vredsum.vs : Vector Integer Sum Reduction

Syntax: `vredsum.vs vd, vs2, vs1, vm`
Behaviour: Computes the wrapping integer sum:
vd[0] = vs1[0] + $\sum$(activeelements of vs2). Overflow wraps modulo 2^{SEW}. The scalar accumulator in `vs1`[0] is always included regardless of masking. If `vl` = 0, no operation is performed. The instruction is used for computing the denominator sum over blocks of elements in the softmax normalisation pass.

```
vmv.s.x   v20, zero               # seed sum accumulator with 0
vredsum.vs v20, v12, v20          # v20[0] = sum(v12[*])
```

3.4.8 Mask, Compare, and Merge Instructions

Mask instructions produce or manipulate bit-packed mask registers. Each mask element is a single bit, located at bit position i in the mask register, independent of SEW or LMUL. Compare instructions compare vector elements against a scalar or another vector and write 1 or 0 into individual mask bits. Merge instructions select between two sources on a per-element basis controlled by a mask register. Together, these instructions form the foundation of data-dependent control flow in the vector domain.
Per the RVV specification, instructions that write a mask register (compare, mask logical) always operate with a *tail-agnostic* policy on the destination mask bits, regardless of the `vta` setting. This applies because mask destination values are always treated as tail-agnostic to reduce the complexity of managing mask data at bit granularity.

vmslt : Vector Signed Less Than

Syntax: `vmslt.vv vd, vs2, vs1, vm` `vmslt.vx vd, vs2, rs1, vm`
Behaviour: Sets mask bit i of `vd` to 1 if vs2[i] < vs1[i] (or `rs1`), treating both as signed values. The `.vi` (immediate) form is not provided for `vmslt` in the RVV specification; comparisons against an immediate using a less-than relation are achieved via `vmsle.vi` with the immediate decremented by one. Prestart mask bits are left undisturbed; tail mask bits are always updated with a tail-agnostic policy.

vmseq : Vector Equal

Syntax: `vmseq.vv vd, vs2, vs1, vm`
`vmseq.vx vd, vs2, rs1, vm` `vmseq.vi vd, vs2, imm, vm`
Behaviour: Sets mask bit i of `vd` to 1 if vs2[i] = vs1[i] (or `rs1`, or sign-extended *imm*). Unlike `vmslt`, the immediate (`.vi`) form is available for equality testing. Tail mask bits always follow the tail-agnostic policy.

vmerge : Conditional Merge

Syntax: `vmerge.vvm vd, vs2, vs1, v0`
`vmerge.vxm vd, vs2, rs1, v0`
`vmerge.vim vd, vs2, imm, v0`
Behaviour: Performs a per-element select over all body elements (from `vstart` to `vl`−1). Per the RVV specification, for each body element i:

$$\text{vd}[i] = \text{v0.mask}[i] \ ? \ \text{vs1}[i] \ : \ \text{vs2}[i]$$

When `v0.mask`[i] = 1, the instruction selects the second source operand (`vs1` or `rs1` or sign-extended *imm*); when `v0.mask`[i] = 0, it selects `vs2`. Unlike regular arithmetic instructions, `vmerge` operates on *all* body elements, it is never further gated by another mask. The mask operand is always fixed to `v0` (the encoding is always `vm` = 0). Tail elements follow `vta`.

```
vmslt.vx v0, v8, a2          # v0[i] = 1 if v8[i] < threshold
vmerge.vxm v12, v12, a3, v0 # v12[i] = (v0[i]) ? a3 : v12[i]
```

vmnot.m : Mask Bitwise NOT

Syntax: `vmnot.m vd, vs2`
Behaviour: Computes the bitwise complement of mask register `vs2`, placing the result in `vd`. Per the RVV specification, this is an assembler pseudoinstruction that expands to `vmnand.mm vd, vs2, vs2`. Mask logical instructions, including `vmnot`, are always unmasked (the `vm` = 0 encoding is reserved). All mask bits, including tail bits beyond the active `vl`, are updated with a tail-agnostic policy.

vmandn.mm : Mask AND-NOT

Syntax: `vmandn.mm vd, vs2, vs1`
Behaviour: Computes vd.mask[i] = vs2.mask[i] AND $\overline{\text{vs1.mask}[i]}$ for each bit. Mask logical instructions are always unmasked; the `vm` = 0 encoding is reserved. Tail mask bits are updated with a tail-agnostic policy. Note that the previous assembler mnemonic for this instruction was `vmandnot`; the RVV specification has since renamed it to `vmandn` for consistency with the scalar ISA. VXT uses the current name.

vid.v : Vector Element Index

Syntax: `vid.v vd, vm`
Behaviour: Writes the element index i (0, 1, 2, ..., vl−1) into each active element of `vd`. Per the RVV specification, masking does not change the index value written to active elements, the value written is always the element's position index, regardless of `v0`. The result is zero-extended to fill SEW-width elements; if the result value would overflow SEW, the least-significant SEW bits are retained (not an issue for VXT since VLMAX ≤ 32 ≤ 255). Prestart elements are left undisturbed; tail elements follow `vta`.

```
vid.v v0                     # v0[i] = i  (element position index)
```

3.5 Known ISA-Level Quirks the Programmer Must Account For

VXT has one documented ISA-level quirk that affects software and must be worked around in any code that uses the e32, m2 configuration. The quirk is presented here purely as a programming constraint and workaround; the root cause in the microarchitecture is explained in Chapter 4.

3.5.1 The e32,m2 floor(vl/2) Erratum

From a programming standpoint, code that requires e32, m2 operation must not rely on all `vl` elements being updated in a single instruction invocation. The correct workaround is to **decompose the e32,m2 operation into two separate e32,m1 passes**, each targeting a distinct register from the pair that would otherwise form the m2 group. Each e32, m1 pass processes up to VLMAX = 4 elements. Concretely:

```
# Workaround for e32,m2 erratum:
# BROKEN -- do not use e32,m2 for arithmetic:
#   vsetvli  t0, a0, e32, m2, ta, ma
#   vadd.vv v16, v16, v8          # only floor(vl/2) elements updated

# CORRECT -- decompose into two e32,m1 passes:
vsetvli  t0, a0, e32, m1, ta, ma  # first 4 elements
vadd.vv  v16, v16, v8              # process elements 0..3
sub      a1, a0, t0                # remaining element count
vsetvli  t1, a1, e32, m1, ta, ma  # next up to 4 elements
vadd.vv  v17, v17, v9              # process elements 4..7 (next registers)
```

Note that the widening instructions `vwmul`, `vwmulu`, and `vwmacc` that *produce* a $2 \times$ SEW result in a register group equivalent to m2 are **not** affected by this erratum; they correctly produce all `vl` results. The erratum applies only to instructions that *consume* an e32, m2 register group as an input or update an e32, m2 destination as a non-widening operation. Any kernel that loads, stores, shifts, adds, or otherwise operates on a 32-bit accumulator at m2 grouping must use the two-pass e32, m1 workaround described above. The root cause of this behaviour is documented in Chapter 4.

3.6 Divergences from the RVV Specification and Justification for Each

VXT is architecturally inspired by and largely compatible with the RISC-V Vector Extension v1.0 (ratified), but it intentionally diverges from that specification in several respects. This section documents every known divergence, describing both the standard RVV behaviour and the VXT behaviour, and explains the justification for each departure. Programmers porting RVV-conforming code to VXT must review this section carefully before assuming standard behaviour.

Table 3.4 summarises all documented divergences. Detailed justifications for each divergence follow in the subsections below.

Table 3.4: VXT divergences from the RVV 1.0 specification

ID	Feature	RVV Behaviour	VXT Behaviour
D-01	`vmax`/`vmin` tail policy	Tail policy governed by `vta`: agnostic mode allows filling tails with 1s or leaving them undisturbed; undisturbed mode always preserves them.	VXT always applies tail-undisturbed policy for `vmax`, `vmin`, `vmaxu`, `vminu` regardless of `vta`.
D-02	Fractional LMUL absent	LMUL values of mf2 (1/2), mf4 (1/4), and mf8 (1/8) are defined.	Fractional LMUL encodings set `vill` = 1. Software must use AVL-based element count reduction via `vsetvli` instead.
D-03	SEW = 64 not supported	e64 is a legal SEW when VLEN ≥ 64.	e64 sets `vill` = 1. Only e8, e16, and e32 are legal.
D-04	LMUL > 2 not supported	m4 and m8 aggregate 4 or 8 registers into a logical vector register group.	m4 and m8 set `vill` = 1. Only m1 and m2 are supported.
D-05	Indexed addressing absent	Indexed (scatter/gather) vector loads and stores access memory at per-element addresses from a second vector register.	Indexed loads and stores are not implemented. Issuing them produces undefined results.
D-06	No floating-point vector instructions	RVV defines a full set of vector floating-point instructions when the F or D scalar extension is present.	No floating-point vector instructions are implemented. VXT is strictly an integer and fixed-point co-processor.
D-07	e32,m2 ⌊vl/2⌋ erratum	An instruction at e32,m2 processes all `vl` elements as specified.	VXT processes only ⌊vl/2⌋ elements at e32,m2. See Section 3.5.1 for the workaround.
D-08	Widening source SEW ≤ 16 only	Widening from e32 to e64 is permitted when VLEN ≥ 64 and e64 is supported.	Widening is only legal when the narrow-domain source SEW is 8 or 16. The e32 → e64 chain is not supported.
D-09	Segment memory operations absent	`vlseg`/`vsseg` load/store interleaved multi-field data structures.	No segment load or store instructions are implemented.
D-10	`vmv.v.v` not implemented	`vmv.v.v vd, vs1` copies a vector register group to `vd` using the current `vl` and `vtype`.	VXT marks `vmv.v.v` as an invalid instruction. The encoding is decoded as `instr_valid` = 0.
D-11	`vsaddu` / `vnclipu` absent	`vsaddu` provides unsigned saturating addition; `vnclipu` provides unsigned narrowing saturating clip.	Only signed variants `vsadd` and `vnclip` are implemented. Unsigned saturating forms are absent.
D-12	`vwmacc.vx` scalar sign-extension broken	The scalar operand `rs1` is sign-extended to SEW width and broadcast to all lanes before the widening multiply-accumulate.	The scalar operand in `vwmacc.vx` is not correctly sign-extended in the VXT hardware. The result is incorrect for all values of `rs1`. Use `vmv.v.x` to broadcast the scalar into a vector register, then use `vwmacc.vv` instead. See Section 3.6.7 for the workaround.

3.6.1 D-01: vmax / vmin / vmaxu / vminu Tail-Undisturbed Policy

The RVV specification permits conforming implementations that declare tail-agnostic (`ta`) mode to write any value, specifically all-ones or the undisturbed previous value, to tail elements during `vmax`, `vmin`, `vmaxu`, and `vminu`. VXT always applies tail-undisturbed policy for these four instructions regardless of the `vta` field. From the programmer's perspective this is a conservative restriction: code written for RVV that does not depend on tail elements receiving any particular value will produce identical results on VXT. Code that relies on tails being filled with all-ones when `ta` is set will see undisturbed values instead; such code must be corrected before targeting VXT.

3.6.2 D-02 through D-06: Structural Exclusions

Divergences D-03 through D-07 are structural exclusions: the VXT hardware simply does not contain the logic for these features. Fractional LMUL, e64, m4/m8, indexed memory, and floating-point are all absent from the physical implementation. Any attempt to use them causes `vill` to be set (for configuration-time exclusions) or produces undefined results (for instruction-time exclusions). Software targeting VXT should begin porting with a static analysis pass to identify any use of excluded features.

3.6.3 D-07: e32,m2 floor(vl/2) Erratum

This divergence is an unintended deviation from the intended behaviour of VXT rather than a deliberate design choice, and it is therefore classified as an erratum rather than a design decision. The two-pass $\mathrm{e32}, \mathrm{m1}$ workaround prescribed in Section 3.5.1 is the only supported mitigation. Code that does not follow the workaround will produce incorrect results for all operations in the (e32, m2) configuration. The root cause is documented in Chapter 4.

3.6.4 D-08: Widening Source SEW $\leq$ 16 Restriction

This restriction is a deliberate design choice driven by the application domain. All benchmark kernels operate in int8 and int16 arithmetic with int32 accumulation; no kernel requires $\mathrm{int32} \times \mathrm{int32} \rightarrow \mathrm{int64}$ widening. The software implication is that any accumulation that would naturally require $\mathrm{e32} \rightarrow \mathrm{e64}$ widening must instead be decomposed into a two-level widening chain ($\mathrm{e8} \rightarrow \mathrm{e16}$, then $\mathrm{e16} \rightarrow \mathrm{e32}$). This adds code complexity in exchange for a significantly simpler hardware multiplier, which reduces area and critical path delay.

3.6.5 D-09: Segment Memory Omission

Segment loads and stores are omitted because the benchmark application suite does not use interleaved multi-field data structures in performance-critical paths. Image data uses planar format and activation tensors follow NHWC or NCHW layouts that do not require field-interleaved access. Code relying on segment instructions must be restructured to use separate plain loads for each field.

3.6.6 D-10 and D-11: Absent Instruction Variants

`vmv.v.v` is absent because it would require a vector-to-vector register copy datapath that adds routing area without benefiting the inference kernels that motivate the design. The `vsaddu` and `vnclipu` unsigned saturating forms are absent because all fixed-point quantisation pipelines in

the benchmark suite use signed arithmetic throughout; unsigned saturation was not required by any kernel during the design phase.

3.6.7 D-12: `vwmacc.vx` Scalar Sign-Extension Erratum

This is an unintended hardware defect rather than a deliberate design choice. The `vwmacc.vx` instruction takes a scalar integer operand in `rs1`, which the RVV specification requires to be sign-extended to SEW width and broadcast to all lanes before the widening multiply-accumulate. The VXT hardware does not perform this sign extension correctly, producing wrong results for all invocations of `vwmacc.vx`.
The workaround is straightforward and adds one instruction:

```
# BROKEN -- do not use vwmacc.vx:
#   vwmacc.vx v16, a0, v8      # scalar a0 not sign-extended correctly

# CORRECT -- broadcast first, then use vwmacc.vv:
vmv.v.x  v10, a0               # broadcast scalar to all lanes of v10
vwmacc.vv v16, v10, v8         # v16 += v10 * v8  (correct result)
```

All convolution, GEMM, and activation kernel code in this project uses `vmv.v.x` + `vwmacc.vv` exclusively. The root cause is documented in Chapter 4.

Summary

This chapter has established the complete and authoritative software-visible contract for the VXT co-processor, verified against the *RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture* (version 1.0, ratified). The supported instruction subset covers configuration (`vsetvli`, `vsetivli`), integer unit-stride and strided memory access, move and broadcast, integer arithmetic, signed and unsigned min/max, shifts, widening and narrowing arithmetic, reductions, and mask operations. The architectural parameters are VLEN = 128 bits, SEW $\in \{8, 16, 32\}$, and LMUL $\in \{\text{m1}, \text{m2}\}$. All instruction semantics, including `vl` constraints, tail and mask agnostic policy, operand overlap rules, and reduction initialisation, follow the RVV specification precisely except where documented as divergences. The VXT_FENCE mandatory programming rule requires the programmer to issue a zero-weight widening multiply-accumulate after every vector ALU instruction before reading the result. The $\text{e32}, \text{m2}$ $\lfloor \text{vl}/2 \rfloor$ erratum constrains the use of that configuration and demands a two-pass $\text{e32}, \text{m1}$ workaround. Twelve deliberate divergences and one hardware erratum (D-12, the `vwmacc.vx` scalar sign-extension defect) from the RVV 1.0 specification have been enumerated and justified.
With this contract established, Chapter 4 proceeds to describe how the hardware satisfies it: the micro-architectural organisation, pipeline structure, memory subsystem, and the internal mechanisms that give rise to both the VXT_FENCE requirement and the $\text{e32}, \text{m2}$ erratum.

Chapter 4

VXT Microarchitecture and Hardware Implementation

Contents

4.1 Introduction 49
4.1.1 Design Goals 49
4.2 System-Level Microarchitecture 49
4.2.1 Top-Level Block Diagram 50
4.2.2 Module Inventory and Responsibilities 50
4.2.3 Data and Control Flow 51
4.2.4 Integration with CV32E40P 52
4.3 Instruction Decoder (`vxt_decoder`) 53
4.3.1 Role and Combinational Nature of the Decoder 53
4.3.2 Block Diagram 54
4.3.3 Port-Level Interfaces 54
4.3.4 `funct3`/`funct6` Decode Logic 56
4.3.5 `vsetvli`/`vsetivli` Configuration Decode 57
4.3.6 Special Cases 57
4.3.7 Design Notes and Known Limitations 58
4.4 CSR Unit (`vxt_csr_rvv`) 59
4.4.1 Role and Update Paths 59
4.4.2 Block Diagram 60
4.4.3 Port-Level Interface 60
4.4.4 Implemented CSRs and Bit-Field Maps 62
4.4.5 VL Computation Logic for `vsetvl`/`vsetivli` 63
4.4.6 Fixed-Point Saturation (`vxsat`) Handling 64
4.4.7 Reset Behaviour and Reset State Summary 65
4.4.8 Timing Considerations 65
4.5 Vector Register File (`vxt_regfile`) 66
4.5.1 Organisation 66
4.5.2 Read/Write Port Structure 67

4.5.3 Write-Timing Considerations . . . 67
4.6 Vector Controller / FSM (vxt_controller) . . . 68
4.6.1 FSM State Overview . . . 68
4.6.2 State Diagram . . . 70
4.6.3 State-by-State Functional Description . . . 72
4.6.4 Instruction Execution Latency . . . 75
4.6.5 Special Write-Enable Paths and Whole-Register-Move Handling . . . 77
4.6.6 Widening Operations and Reduction Handling . . . 77
4.6.7 VTA/VMA Policy Enforcement at the Controller Level . . . 78
4.7 Vector Load/Store Unit (vxt_vlsu) . . . 79
4.7.1 Overview and OBI-Based Transaction Model . . . 79
4.7.2 Block Diagram . . . 80
4.7.3 Byte-Enable / Active-Element Logic (elem_active_w, Masking, Prestart/Tail Handling) . . . 81
4.7.4 Address Generation and Transaction-Valid Logic . . . 82
4.7.5 Store Path (Pass-Through, Byte-Enable Restriction) . . . 83
4.7.6 Strided Load Logic and Window-Batching Optimisation . . . 83
4.7.7 Unit-Stride Load Merge Logic (4-Way Priority Mux) . . . 84
4.7.8 Sequencer FSM (busy_q, req_cnt_q, pack_cnt_q, rdata_q) and OBI Handshake Timing . . . 84
4.7.9 LMUL=2 Multi-Beat Operation at the VLSU Level . . . 85
4.8 Vector ALU (vxt_alu) . . . 86
4.8.1 Overview . . . 87
4.8.2 Block Diagram . . . 88
4.8.3 Port-Level Interfaces . . . 88
4.8.4 Operand-B Multiplexer (op_b_mux) . . . 90
4.8.5 Main Arithmetic Datapath (vxt_alu Top-Level) . . . 91
4.8.6 Multiplier Sub-Module (vxt_mul) . . . 93
4.8.7 Reduction Tree (vxt_reduction_tree) . . . 93
4.8.8 Design Notes and Known Limitations . . . 94
4.9 RVV Policy Implementation Across the Datapath . . . 94
4.9.1 vstart: Prestart Elements . . . 95
4.9.2 vta: Tail Agnostic . . . 95
4.9.3 vma: Mask Agnostic . . . 96
4.9.4 Mask Register Indexing Across LMUL Beats . . . 96
4.10 Arbitration and Bus Integration . . . 96
4.10.1 Why Arbitration Is Needed . . . 96
4.10.2 Arbitration Scheme . . . 97
4.10.3 Timing Stability Guarantees . . . 97
4.11 Key Design Decisions and Trade-offs . . . 98
4.12 Summary . . . 99

4.1 Introduction

The preceding chapters have built the context and the software-visible contract that this chapter now fulfills in hardware. Chapter 2 surveyed the broader landscape of RISC-V vector processing and embedded AI acceleration, positioning VXT as a constrained, non-pipelined RVV subset designed for low-area FPGA deployment rather than full specification compliance. Chapter 3 then defined the architectural contract precisely: the supported instruction subset, the VLEN/SEW/LMUL support matrix, instruction semantics by class, and the programmer-visible rules, including the mandatory `VXT_FENCE` contract and the `e32m2 floor(vl/2)` erratum, that software must observe when targeting this implementation. This chapter shifts focus from specification to realization, and addresses *how* those instructions are actually executed in hardware.

Specifically, this chapter presents the microarchitecture of VXT: the internal organization of functional blocks, the control logic that sequences instruction execution, and the datapaths that move data between the vector register file, the arithmetic units, and memory. Where Chapter 3 described instruction semantics at the level of an ISA, this chapter describes the register-transfer-level (RTL) structures in SystemVerilog that implement those semantics. It also provides the hardware-level explanations for the two programmer-visible constraints stated without justification in Chapter 3: the `VXT_FENCE` requirement follows directly from the non-pipelined, write-back-on-retire execution model described in Section 4.6, and the `e32m2 floor(vl/2)` erratum is a consequence of the LMUL=2 element-count derivation logic in the controller FSM, analyzed in Section 4.6.

4.1.1 Design Goals

Three design goals shaped every microarchitectural decision described in this chapter:

- **Non-pipelined execution.** VXT processes one vector instruction to completion before accepting the next. This decision trades peak throughput for a substantially simpler control unit: hazard detection, forwarding logic, and pipeline interlocks that a pipelined vector unit would require are eliminated entirely. For a graduation-project-scale design with a single execution lane, the throughput cost of this choice is modest relative to the verification and design-time savings it provides.
- **Area efficiency.** VXT is designed to be implementable on resource-constrained FPGA targets without requiring register file duplication, multiple parallel ALU lanes, or deep pipeline buffering typical of high-performance vector units. Functional blocks are reused across instruction types wherever possible, for example, a single ALU datapath serves both single-width and widening operations, and control state is kept minimal.
- **Drop-in coprocessor integration.** VXT is not a standalone core; it attaches to an existing CV32E40P scalar pipeline with minimal modification to the host core. This required defining a narrow, well-specified interface (Section 4.2.4) through which the scalar core detects vector instructions, asserts a stall, and hands off execution to VXT, while remaining otherwise unaware of VXT's internal structure.

4.2 System-Level Microarchitecture

Before describing individual functional blocks in detail, this section establishes the full system picture: how the blocks are arranged, what each one is responsible for, and how data and control

signals flow through the design from instruction fetch to result writeback. This top-down view is the reference frame for all subsequent sections.

4.2.1 Top-Level Block Diagram

Figure 4.1 shows the top-level organisation of the VXT subsystem. At its boundary, VXT presents a single interface to the CV32E40P scalar core and a pair of OBI master ports to the shared memory bus, arbitrated by the `obi_mux` module.

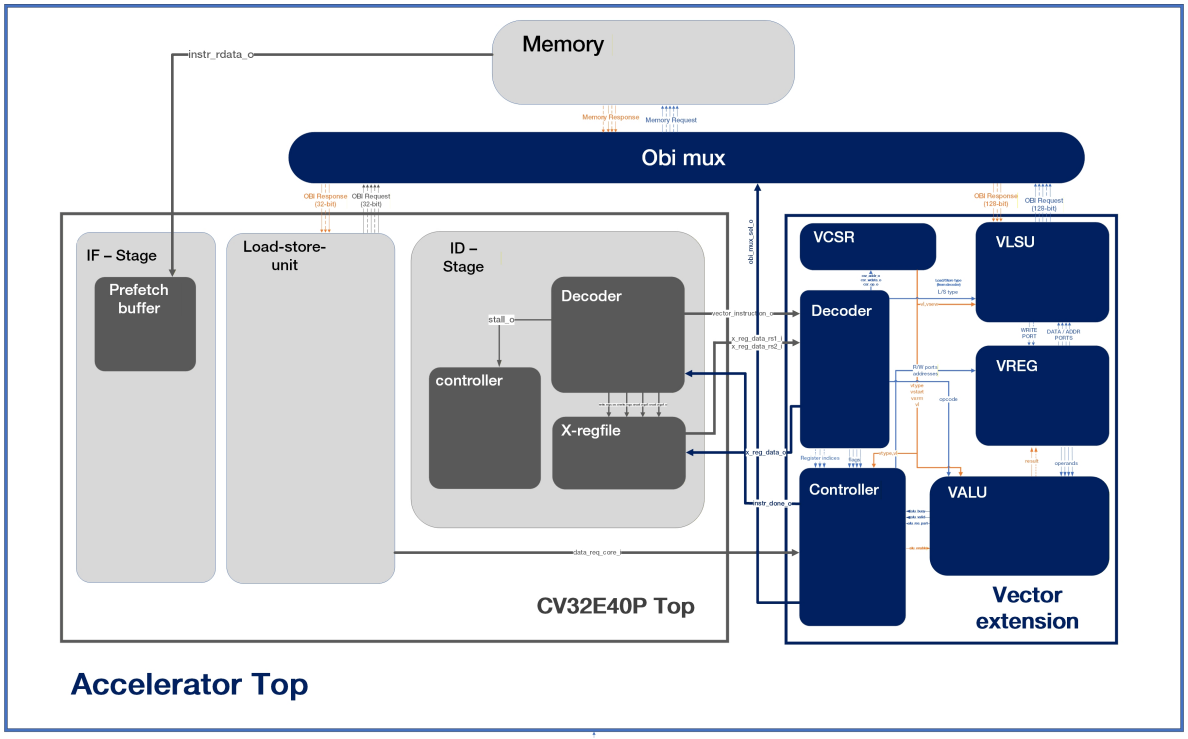

Figure 4.1: Top-level block diagram of the VXT subsystem. Solid arrows represent data paths; dashed arrows represent control signals. The `obi_mux` arbitrates shared memory access between the scalar core and the VLSU.

4.2.2 Module Inventory and Responsibilities

The following paragraphs summarise the purpose of each RTL module; detailed treatment is deferred to the section cited in each entry.

`vxt_top`. The structural top-level wrapper. It instantiates all subordinate modules, routes inter-module signals, and exposes the external port list consumed by the CV32E40P integration layer. It contains no combinational logic of its own beyond port multiplexing.

`vxt_decoder` **(Section 4.3).** A purely combinational block that decodes the `opcode`, `funct3`, and `funct6` fields of the incoming 32-bit instruction. It produces an enumerated operation code, source and destination register addresses, immediate values, operand-select signals, and a set of flags that the controller consumes to determine execution mode. The decoder is re-evaluated every cycle but its outputs are only acted upon when the controller is in the appropriate state.

`vxt_controller` **(Section 4.6).** The sequencing heart of VXT. Its FSM transitions through IDLE, DECODE, ALU_EXEC, MEM_EXEC, and RETIRE states, asserting write-enable,

transaction-valid, and done signals at the right cycle. Multi-beat operations (LMUL=2, widening, reductions) are managed entirely within this FSM without exposing additional complexity to the decoder or register file.

`vxt_csr_rvv` **(Section 4.4).** Implements the `vsetvli`/`vsetivli` semantics and stores the five vector CSRs mandated by the RVV specification (`vl`, `vtype`, `vstart`, `vxsat`, `vcsr`). It exports `vl`, `vsew`, `vlmul`, and the two policy bits `vta`/`vma` to the rest of the datapath on every cycle.

`vxt_regfile` **(Section 4.5).** A synchronous register file holding 32 vector registers, each `VLEN` bits wide. It provides two read ports (for `vs1` and `vs2`) and one write port (for `vd`), with the write being clocked on the negative edge to satisfy the same-cycle read-after-write requirement of the non-pipelined execution model.

`vxt_alu` **(Section 4.8).** Implements the element-wise compute datapath. A generate loop instantiates one arithmetic slice per supported element, with each slice capable of 8-, 16-, and 32-bit operations controlled by the decoded `vsew` value. Widening results are written to a double-width accumulator buffer managed by the controller.

`vxt_vlsu` **(Section 4.7).** Translates a single `vle.v`/`vse.v`/`vlse.v` instruction into a sequence of OBI read or write transactions. An internal sequencer FSM tracks outstanding transactions, assembles returning read data into the correct register-file word positions, and signals the controller when all elements have been transferred.

`obi_mux` **(Section 4.10).** A two-to-one priority arbiter that interleaves the VLSU's OBI requests with those of the scalar core's data interface. When VXT is idle the scalar core has uncontested bus access; when a vector load or store is in progress the arbiter grants the bus to the VLSU and back-pressures the scalar core via the OBI `gnt` signal.

4.2.3 Data and Control Flow

Execution of a vector instruction proceeds through five conceptual stages, each corresponding to one or more FSM states in the controller.

1. **Instruction acceptance.** The CV32E40P decoder recognises a vector opcode (`0x57`). VXT asserts `vxt_stall` to freeze the scalar pipeline and captures the instruction word, `rs1`, and `rs2` values from the scalar register file into local registers.

2. **Decode.** The controller enters DECODE for one cycle, during which the combinational decoder resolves the operation type, operand addresses, and immediate fields. Configuration instructions (`vsetvli`/`vsetivli`) complete here without further execution stages: the CSR unit updates `vl` and `vtype` and the controller moves directly to RETIRE.

3. **Dispatch.** The controller selects the appropriate execution path, ALU_EXEC for arithmetic, logical, shift, widening, and reduction operations, or MEM_EXEC for loads and stores, and asserts the corresponding enable signals to the ALU or VLSU.

4. **Execute.** For ALU instructions, the register file supplies operands on the same cycle the controller enters ALU_EXEC; the ALU produces a result that is written back to the register file at the end of that cycle. Multi-cycle operations (LMUL=2 or multiply) iterate within ALU_EXEC under controller management. For memory instructions, the

VLSU issues OBI transactions over one or more cycles until all active elements have been transferred, then asserts `vlsu_done` to the controller.

5. **Retire.** The controller enters RETIRE, de-asserts `vxt_stall`, and signals completion to the scalar core, which then resumes normal pipeline operation.

4.2.4 Integration with CV32E40P

CV32E40P is a four-stage in-order scalar core (IF, ID, EX, WB) with no native support for vector instructions. Integrating VXT required a minimal but carefully chosen set of interface signals that intercept vector instructions at the ID/EX boundary without disturbing the scalar pipeline's existing control logic.

Interface Signals

Table 4.1 lists the signals that cross the boundary between CV32E40P and VXT.

Signal	Direction	Description
`vxt_instr`	Core → VXT	32-bit instruction word captured at the ID stage.
`vxt_rs1`	Core → VXT	Value of scalar register `rs1`, used by `vsetvli`, `vmv.s.x`, and strided-address computation.
`vxt_rs2`	Core → VXT	Value of scalar register `rs2`, used as the stride operand in strided load/store instructions.
`vxt_data_req`	Core → VXT	Indicates a scalar load/store request and is used by the VXT to insert a one-cycle delay before accessing the shared memory.
`vxt_done`	VXT → Core	Asserted by the VXT when the current vector instruction completes, allowing the scalar core to continue execution.
`vxt_rd_data`	VXT → Core	Scalar result written back to the integer register file (used by `vmv.x.s` and `vsetvli`).

Table 4.1: Interface signals between CV32E40P and the VXT coprocessor.

Handshake Mechanism

The stall-based handshake is simple by design to minimise changes to the host core. When the CV32E40P ID stage decodes an instruction with opcode `0x57`, it forwards the instruction word together with the current `rs1` and `rs2` values to the VXT. The VXT then asserts `stall_i`, causing the decode stage to remain in the DECODE state. While `stall_i` is asserted, `is_decoding_o` is deasserted and the control FSM holds `ctrl_fsm_ns` in DECODE, preventing the processor from advancing to the next instruction. The stall may last from a single cycle for configuration instructions to many cycles for long-latency vector operations, including multi-element memory accesses. Once the vector operation completes, the VXT deasserts `stall_i`, allowing the control FSM to resume normal execution. From the scalar core's perspective, a

vector instruction therefore behaves like any other long-latency operation, such as a multiply or divide: execution pauses while the instruction is processed and resumes transparently once the stall is released.

4.3 Instruction Decoder (`vxt_decoder`)

The instruction decoder is the entry point through which every vector instruction enters the VXT datapath. This section describes its combinational structure, its interfaces to the surrounding blocks, the decode logic it implements for each instruction class, and the special-case handling required to resolve aliasing within the RVV encoding space.

4.3.1 Role and Combinational Nature of the Decoder

`vxt_decoder` is a purely combinational module: it contains no registers and no internal state, and its outputs are pure functions of the current instruction word and a small number of configuration inputs from the CSR unit (`vsew`, `vlmul`). This design choice follows directly from the non-pipelined execution model established in Section 4.2, since only one instruction is ever in flight, there is no benefit to registering decode results ahead of time, and keeping the decoder combinational avoids an entire class of timing hazards that a registered decode stage would introduce (e.g. the decoder producing stale outputs for one cycle after a new instruction arrives).

Functionally, the decoder's role is to translate the flat 32-bit RVV instruction encoding into a structured, named set of control and data signals that the rest of VXT can consume without any further awareness of instruction encoding. Every other block in the design (the controller, the ALU, the CSR unit, the register file) operates exclusively on the decoder's outputs and never inspects the raw instruction word directly. This separation of concerns means the entire instruction encoding (and any future extension to it) is localized to a single module.

Because the decoder is combinational, its outputs are valid every cycle that a valid instruction word is present at its input, regardless of which state the controller FSM currently occupies. It is the controller's responsibility (Section 4.6) to determine *when* the decoder's outputs should be acted upon; the decoder itself makes no judgment about timing.

4.3.2 Block Diagram

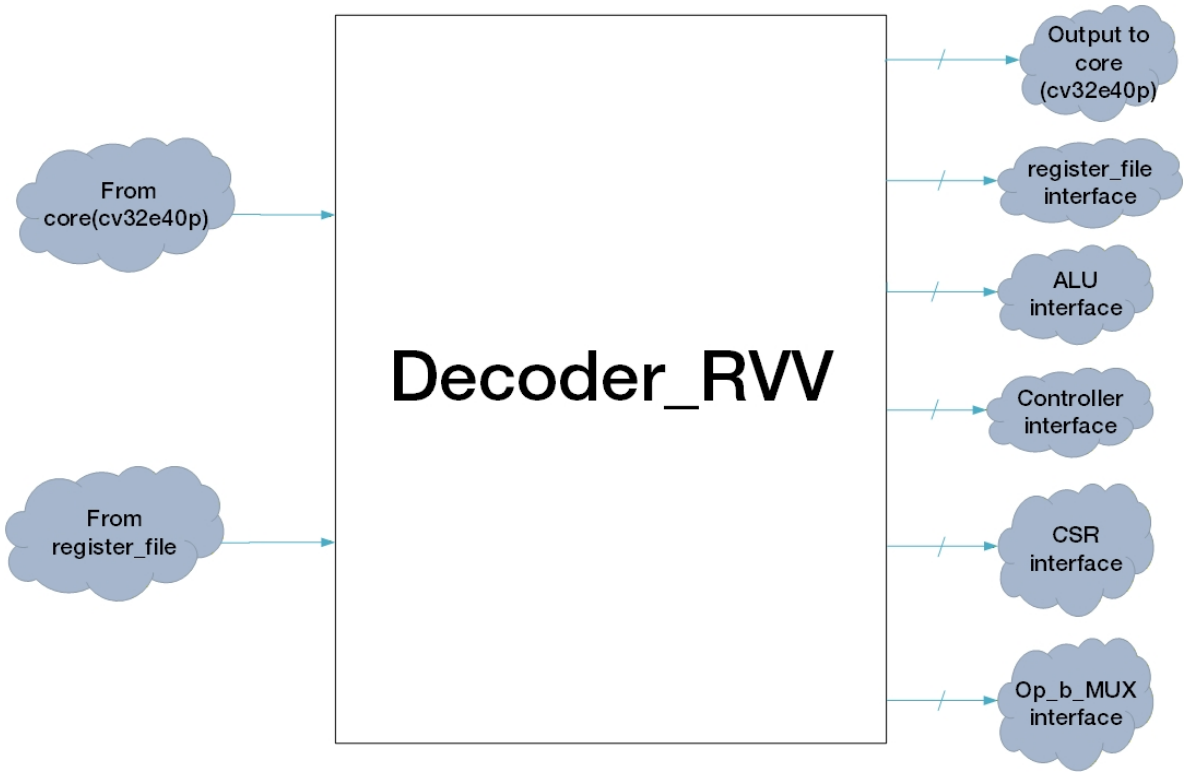

Figure 4.2: Internal combinational block diagram of the `vxt_decoder` module, detailing field extraction and control signal generation.

4.3.3 Port-Level Interfaces

The decoder sits at the centre of the VXT block diagram (Figure 4.1) and exposes five distinct interface groups, summarised in Table 4.2.

Table 4.2: Port-level interfaces of `vxt_decoder`, grouped by destination block.

Group	Signal	Dir	Description
Scalar core interface			
	`instr_i[31:0]`	In	32-bit instruction word from the CV32E40P ID stage.
	`x_reg_data_rs1_i`, `x_reg_data_rs2_i`	In	Integer register values for `rs1`/`rs2`, forwarded from the scalar register file.
	`x_reg_data_o`	Out	Scalar writeback value: `vmv.x.s` sign-extended result; or new `vl` from `vsetvli`/`vsetivli`.
	`x_reg_data_rs1_o`, `x_reg_data_rs2_o`	Out	Pass-through of `rs1`/`rs2` to the VLSU for base-address and stride operands.
Register file interface			
	`vs1_o`, `vs2_o`, `vd_o` [4:0]	Out	Decoded 5-bit register addresses from `instr_i[19:15]`, `[24:20]`, and `[11:7]`.

Continuation of Table 4.2			
Group	**Signal**	**Dir**	**Description**
	operand_c_i [VLEN-1:0]	In	Current vd contents, fed back for VMV and tail/mask policy inside w_data_o.
	v_reg_data_i [VLEN-1:0]	In	vs2 contents; used by vmv.x.s to produce the sign-extended scalar result.
	w_data_o [VLEN-1:0]	Out	Pre-computed write-data for VMV instructions, with tail and mask policy already applied.
ALU interface			
	op_o (alu_op_e)	Out	Enumerated ALU operation code (e.g. ALU_VADD, ALU_VWMUL).
	unit_o, mode_o	Out	Execution unit selector and arithmetic mode (MODE_ARITH, MODE_WIDEN, MODE_MOVE).
	operand_o (operand_e)	Out	Operand form: IVV, IVX, IVI, MVV, or MVX.
	widen_o, narrow_o	Out	Asserted for widening and narrowing instructions respectively.
	mask_comp_o, mask_logic_o	Out	Distinguish mask-comparison (vmslt, vmseq) from mask-logic (vmandn, vmnand) operations.
	vm_o	Out	Mask-enable bit (active-low of instr[25]); when clear, all elements are written.
	funct3_o[2:0], imm5_o[4:0]	Out	Raw funct3 field and 5-bit immediate, forwarded to the operand-B mux.
	use_imm_o	Out	Asserted for OPIVI-form instructions to select imm5_o as the second ALU operand.
Controller interface			
	is_load_o, is_store_o	Out	Memory-class classifiers; route the instruction to the VLSU path.
	is_config_o	Out	Asserted for vsetvli/vsetivli; fast-paths directly to RETIRE.
	is_mul_o	Out	Asserted for vmul, vmulh, vwmul, vwmacc; enables multi-cycle sequencing.
	is_reduction_o	Out	Asserted for vredmax.vs and vredsum.vs.
	is_vmv_o, is_vmovnr_o, is_vmv_xs_o, is_vmv_imm_o	Out	Move-class sub-type flags for vmv.v.*, whole-register move, vmv.x.s, and immediate-broadcast forms.

Continuation of Table 4.2			
Group	**Signal**	**Dir**	**Description**
	`is_csr_o`	Out	Asserted for Zicsr instructions (`csrrw`/`csrrs`/`csrrc` and immediate forms).
	`instr_valid_o`	Out	Asserted when the instruction matches a supported encoding.
CSR unit interface			
	`vtypei_o`[8:0]	Out	Extracted `zimm` field (`vsew`, `vlmul`, `vta`, `vma`) from `vsetvli`/`vsetivli`.
	`avl_o`[31:0]	Out	Application vector length: `rs1` value for `vsetvli`; zero-extended 5-bit immediate for `vsetivli`.
	`write_vl_o`, `write_vtype_o`	Out	Write-enable strobes for the `vl` and `vtype` CSRs.
	`vset_op_o`	Out	Distinguishes `VSETVLI` from `VSETIVLI` for the AVL computation.
	`vsew_i`, `vl_i`, `vl_computed_i`, `vstart_i`, `vta_i`	In	Live CSR state fed back for policy application inside `w_data_o`.
Zicsr interface			
	`csr_addr_o`	Out	12-bit CSR address from `instr[31:20]`.
	`csr_wdata_o`[XLEN-1:0]	Out	Write data: `rs1` for register forms; zero-extended `uimm[4:0]` for immediate forms.
	`csr_op_o`	Out	Decoded operation: `READ`, `WRITE`, `SET`, or `CLEAR`.
	`csr_rdata_i`[XLEN-1:0]	In	CSR read-data returned to `x_reg_data_o` for `rd` writeback.
VLSU interface			
	`mop_o`[1:0]	Out	Memory-addressing mode: unit-stride, strided, or indexed (from `instr[27:26]`).
	`width_o`[2:0]	Out	Element width for memory access (from `instr[14:12]`).
	`nf_o`[2:0]	Out	Number-of-fields for segmented load/store (from `instr[31:29]`).

The `op_b_mux` interface deserves particular note: rather than having every downstream consumer of the second ALU operand re-derive which source to use, the decoder resolves this once and exports a single select signal, keeping the operand-selection logic in exactly one place.

4.3.4 `funct3`/`funct6` Decode Logic

RVV arithmetic instructions are encoded with two overlapping fields that together determine the operation to perform: `funct3`, which selects the *operand type* (vector-vector, vector-scalar, vector-immediate, or the distinct OPIVI/OPMVV variants), and `funct6`, which selects the *operation* itself (add, subtract, shift, and so on) independently of operand type.

The decoder treats these two fields as orthogonal decode axes. funct3 is decoded first into one of four operand-source categories (Table 4.3), which directly drives op_b_sel_o. funct6 is decoded in parallel into the enumerated alu_op_o value consumed by the ALU. Because the two fields are independent, the same funct6 encoding produces the same arithmetic operation regardless of which operand form is selected, for example, funct6 = 000000 (ADD) maps to ALU_OP_ADD whether the instruction is vadd.vv, vadd.vx, or vadd.vi.

funct3	Operand category	Second operand source
000	OPIVV	vs2 (vector register)
011	OPIVI	Sign- or zero-extended 5-bit immediate
100	OPIVX	Scalar register rs1
010	OPMVV	vs2 (vector register, integer-multiply class)
110	OPMVX	Scalar register rs1 (integer-multiply class)

Table 4.3: funct3 operand-category decode.

This separation of operand-type decode from operation decode keeps the ALU operation table compact: instead of enumerating every (operand-type, operation) pair as a distinct case, the ALU need only implement each funct6 operation once, and the operand multiplexer (Section 4.3.3) handles routing the correct second operand into it.

4.3.5 vsetvli/vsetivli Configuration Decode

Configuration instructions are decoded by a dedicated branch of combinational logic, since their encoding does not follow the funct3/funct6 pattern used by arithmetic instructions. The decoder distinguishes vsetvli from vsetivli using bit 31 of the instruction word, and from vsetvl (register-register form) using bits 31:30. For both immediate forms, the 11-bit zimm field encoding vtype (vsew, vlmul, vta, vma) is extracted directly from the instruction's immediate position and exported as zimm_o for the CSR unit to interpret.
Two additional single-bit flags are decoded to disambiguate the four vsetvli addressing-mode variants defined by the RVV specification: rs1_is_x0_o, asserted when the source register specifier equals x0, and rd_is_x0_o, asserted when the destination register specifier equals x0. These two flags together select between the requested-vl write to vl (normal case), the maximum-vl case (rs1 = x0, rd $\neq$ x0), and the vl-unchanged case (rs1 = x0, rd = x0), as specified in Chapter 3. The decoder itself does not compute the new value of vl; it only extracts and forwards the fields needed for the CSR unit to perform that computation, keeping the configuration arithmetic localized to vxt_csr_rvv (Section 4.4).

4.3.6 Special Cases

A small number of instructions do not fit cleanly into the regular funct3/funct6 decode structure and required dedicated handling in the decoder.

CSR Instruction Decode (Zicsr)

The decoder also includes dedicated handling for the Zicsr extension, since CSR instructions do not follow the standard RVV instruction decoding based on the funct3 and funct6 fields. Instead, they use a dedicated opcode together with the funct3 field to encode the CSR operation.

When the CSR opcode is detected, the decoder asserts `is_csr_o`, extracts the 12-bit CSR address from instruction bits `[31:20]`, and forwards the current CSR value to the destination register through `x_reg_data_o`. The write operand is selected according to the instruction format. Register-based instructions (`CSRRW`, `CSRRS`, and `CSRRC`) use the value read from the integer register file (`rs1`), whereas immediate variants (`CSRRWI`, `CSRRSI`, and `CSRRCI`) use the zero-extended 5-bit immediate field.
The CSR operation is determined from `funct3[1:0]`, where the decoder generates one of four internal operations: `CSR_OP_WRITE`, `CSR_OP_SET`, `CSR_OP_CLEAR`, or `CSR_OP_READ`. For the set and clear instructions, the decoder follows the RISC-V Zicsr specification by treating a zero `rs1` (or zero immediate in the immediate variants) as a read-only operation, thereby avoiding unnecessary modifications to the addressed CSR. Any invalid `funct3` encoding is marked as an illegal instruction by deasserting the instruction valid signal.

VMV Disambiguation

The OPIVX/OPMVX encoding space is reused by four semantically distinct move/broadcast instructions, `vmv.v.x`, `vmv.s.x`, `vmv.x.s`, and the whole-register move family `vmv⟨nr⟩r.v`, which are not separable by `funct6` alone. The decoder resolves this by additionally inspecting the `vs2` field: a `vs2` encoding of `00000` combined with the relevant `funct6`/`funct3` pattern identifies the scalar-move forms (`vmv.s.x`, `vmv.x.s`), while a non-zero, fixed `vs2` pattern combined with a distinguishing immediate field identifies the register-group-move forms (`vmv1r.v` through `vmv8r.v`). Each of these four forms produces a different combination of classification flags and a different `op_b_sel_o` value, since each moves data along a different path: scalar-to-vector, vector-to-scalar, or vector-to-vector with no element-wise computation at all.

VRSUB Operand Swap

`vrsub.vx`/`vrsub.vi` compute `vd = rs1 - vs2` (or `imm - vs2`), the reverse operand order of every other OPIVX/OPIVI arithmetic instruction, which compute `vd = vs2` *op* `rs1`. Rather than adding a second subtraction opcode to the ALU, the decoder handles this by asserting a one-bit `op_swap_o` flag whenever `funct6` matches the VRSUB encoding. The ALU's operand multiplexer consumes this flag to swap its two inputs ahead of the subtraction, so the ALU itself implements only a single, direction-agnostic subtract operation.

Dual-Encoded `funct6` Values

Several `funct6` bit patterns are reused across more than one instruction, differentiated only by `funct3` or by a field that is an opcode extension in one operand form and a genuine operand in another. For example, the `funct6` pattern used for narrowing shift-right instructions overlaps with the pattern used for a mask-logic operation when the operand category differs. The decoder resolves every such overlap by decoding `funct3` *before* branching on `funct6` within each operand-category branch, ensuring no `funct6` pattern is interpreted outside the operand-category context for which it was assigned. This ordering (category first, operation second) is the same principle that underlies the general `funct3`/`funct6` separation described in Section 4.3.4, applied here to resolve encoding-space reuse rather than just to route operands.

4.3.7 Design Notes and Known Limitations

The decoder's fully combinational structure was a trade-off favouring simplicity and verifiability over timing margin. Because the decode logic forms a single combinational cone feeding directly into the ALU operand multiplexers in the same cycle (Section 4.2.3), the critical path through

vxt_decoder contributes directly to the achievable clock frequency of the design; this was judged an acceptable trade-off given the non-pipelined, area-constrained design goals stated in Section 4.1, but would need to be revisited, most likely by registering decode outputs and accepting an additional pipeline stage, if VXT were extended toward a higher-frequency or pipelined implementation.

A second limitation is that the decoder performs no illegal-instruction detection beyond what is implied by its case-statement structure: an instruction word that does not match any recognised encoding simply produces don't-care or default outputs rather than an explicit illegal-instruction signal. This was an accepted scope reduction for the project, since the toolchain used to generate test binaries (Chapter 3) is trusted to emit only well-formed VXT instructions; a production implementation would require an explicit illegal_instr_o output and corresponding trap-handling support in the scalar core, neither of which exists in the current CV32E40P integration.

Finally, the special-case handling described in Section 4.3.6 (particularly the VMV disambiguation logic) is the single largest contributor to the decoder's logic depth, since it requires inspecting multiple instruction fields simultaneously rather than following the uniform funct3-then-funct6 decode tree used elsewhere. Any future extension of the supported instruction subset that introduces additional encoding-space reuse should be expected to extend this same case-by-case resolution pattern, and is the most likely source of decode-logic complexity growth going forward.

4.4 CSR Unit (vxt_csr_rvv)

The CSR unit is the single repository of vector configuration and status state in VXT. It implements the set of Control and Status Registers mandated by the RVV specification, performs the vsetvli/vsetivli configuration arithmetic referenced but not detailed in Section 4.3.5, and arbitrates three independent update paths against a single register bank: ordinary Zicsr instructions, vector-length-setting instructions, and autonomous status updates driven by the ALU's fixed-point datapath. This section describes the module's interfaces, its register map, and the combinational logic that implements each of these update paths.

4.4.1 Role and Update Paths

vxt_csr_rvv provides a single, self-contained register bank that tracks the active vector configuration and status for the datapath. Three distinct sources may modify this state on any given cycle, and the module's combinational logic determines which one takes effect:

- **Standard CSR instructions** (CSRRW/CSRRS/CSRRC/CSRR), decoded by vxt_decoder (Section 4.3.6) and applied at RETIRE under control of the vxt_controller FSM.
- **Vector-length-setting instructions** (vsetvli/vsetivli/vsetvl), which perform an atomic update of vtype and vl in a single cycle.
- **Autonomous ALU-driven status updates**, specifically the sticky-OR update of vxsat whenever the fixed-point datapath signals a saturation event.

On reset, vill is asserted and all other configuration fields are cleared, placing the core in a defined illegal-vtype state until software explicitly establishes a valid vector context via vsetvli or vsetivli. This deliberate reset-to-illegal policy prevents any vector instruction from being interpreted against a stale or default configuration before software has configured one.

4.4.2 Block Diagram

Figure 4.3 shows the internal organisation of `vxt_csr_rvv`. Three write-data sources, the standard-CSR write-data mux, the `vsetvl` configuration-decode logic, and the `vxsat` sticky-set path, converge on the register bank through a single priority-ordered next-state multiplexer, described in Section 4.4.8. A separate, fully combinational read path serves `csr_rdata_o` directly from the current register contents, independent of which (if any) write path is active in the same cycle.

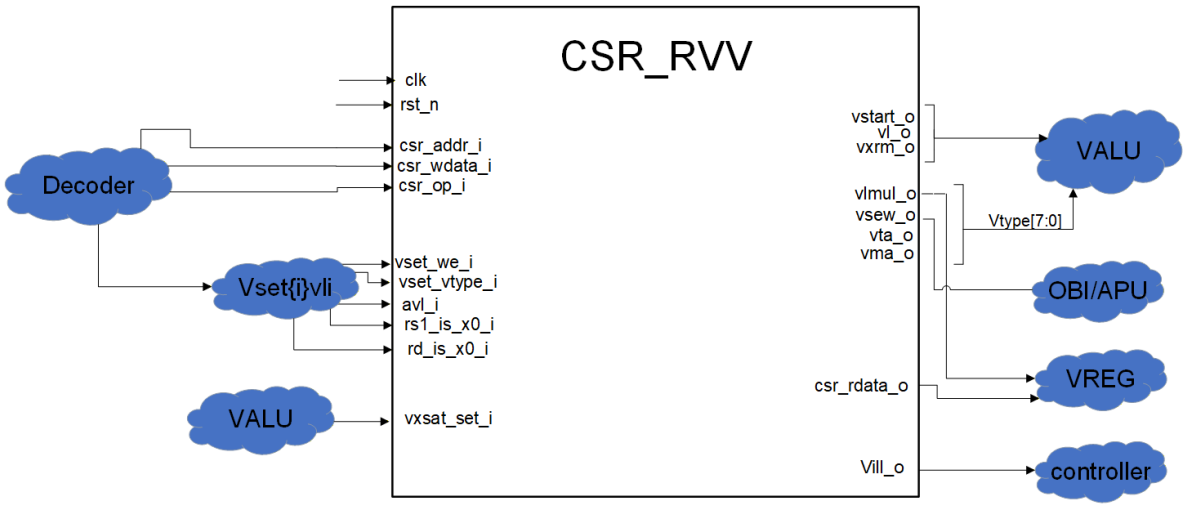

Figure 4.3: Internal block diagram of `vxt_csr_rvv`, showing the three write-data sources (standard CSR, `vsetvl` configuration decode, ALU saturation) arbitrated onto the register bank, and the combinational read-data path.

4.4.3 Port-Level Interface

The CSR unit's ports are grouped into six interfaces, corresponding to the distinct blocks it communicates with. Table 4.4 summarises each group.

Interface	Direction	Description
Global	Input	`clk`, the system clock against which all state registers update on the rising edge, and `rst_n`, an active-low asynchronous reset that clears all registers to their reset values.
Standard CSR (decoder)	Input/Output	`csr_addr_i` (12-bit logical CSR address), `csr_wdata_i` (write data, sourced from `rs1` or a zero-extended immediate), `csr_op_i` (one of `CSR_OP_WRITE`/`SET`/`CLEAR`/`READ`), `csr_we_i` (asserted by the controller only at RETIRE when the current instruction is a CSR instruction), and `csr_rdata_o` (combinationally valid read data).
Vector controller	Output	`vill_o`, asserted whenever the current `vtype` configuration is illegal per the RVV specification; consumed by the controller to gate execution of subsequent vector instructions.
Vector-length setting	Input	`vset_we_i` (one-cycle write enable for `vsetvli`/`vsetivli`/`vsetvl`), `vset_vtype_i` (the 8-bit encoded `vtype` field, `[2:0]=vlmul`, `[5:3]=vsew`, `[6]=vta`, `[7]=vma`), `avl_i` (the application vector length, from `rs1` or a zero-extended 5-bit immediate), `rs1_is_x0_i` and `rd_is_x0_i` (the disambiguation flags forwarded unchanged from the decoder, Section 4.3.5).
Vector ALU	Input/Output	`vxsat_set_i`, asserted by the ALU's fixed-point datapath on a saturation event, and `vxrm_o`, the two-bit fixed-point rounding mode delivered to the ALU.
Configuration outputs	Output	`vlmul_o`, `vsew_o`, `vta_o`, `vma_o`, `vl_o`, and `vstart_o`, the live configuration state exported to every other datapath block on every cycle, regardless of controller state.

Table 4.4: Port-level interfaces of `vxt_csr_rvv`.

The configuration outputs are the most heavily consumed signals in the entire design: the decoder uses `vsew_o`/`vlmul_o` to size operand width and element count, the ALU uses `vsew_o` to select its 8/16/32-bit datapath slice, the controller uses `vl_o` to bound its element-iteration loop, and the register file's masking and tail logic (Section 4.9) consumes `vta_o`/`vma_o` directly. Because these outputs are combinational functions of registered state, they are valid every cycle without additional latency.

4.4.4 Implemented CSRs and Bit-Field Maps

Table 4.5 lists the seven CSRs implemented by `vxt_csr_rvv`, the standard RVV addresses they occupy [2], and their access permissions.

CSR	Address	Access	Description
`vstart`	`0x008`	R/W	First active element index. Cleared to 0 at the start of each `vset*` instruction.
`vxsat`	`0x009`	R/W	Fixed-point saturation flag. Set by the ALU; clearable by software.
`vxrm`	`0x00A`	R/W	Fixed-point rounding mode (2-bit): `00`=rnu, `01`=rne, `10`=rdn, `11`=rod.
`vcsr`	`0x00F`	R/W	Combined view of `vxrm[2:1]` and `vxsat[0]`. A write updates both underlying fields.
`vl`	`0xC20`	R/O	Current vector length set by the most recently executed `vsetvl`/`vsetivli`.
`vtype`	`0xC21`	R/O	Current vector type: `vill[31]`, `vma[7]`, `vta[6]`, `vsew[5:3]`, `vlmul[2:0]`.
`vlenb`	`0xC22`	R/O	Constant `VLEN/8`; the byte length of a single vector register.

Table 4.5: CSRs implemented by `vxt_csr_rvv`.

`vl` and `vtype` are read-only to standard CSR instructions; any `CSRRW`/`CSRRS`/`CSRRC` write attempt targeting either address is silently ignored, since the only architecturally sanctioned way to modify them is through `vsetvli`/`vsetivli`/`vsetvl` (Section 4.4.5). The CSR operation decoder otherwise implements the four standard Zicsr operations directly: `CSR_OP_WRITE` replaces the addressed CSR with the full write-data value, `CSR_OP_SET` performs `csr |= wdata`, `CSR_OP_CLEAR` performs `csr &= ~wdata`, and `CSR_OP_READ` asserts no write at all, leaving the addressed CSR unchanged.

`vtype` Bit-Field Map

Table 4.6 gives the bit-field layout of `vtype` [2] (`0xC21`), a read-only register synthesised combinationally from the underlying configuration state rather than stored as a separate physical register.

Bits	Field	Description
[31]	vill	Illegal-vtype flag. Set when the encoded configuration is not supported.
[30:8]	(rsvd)	Reserved, reads as zero.
[7]	vma	Vector Mask Agnostic policy. 1 = mask-off elements may be overwritten.
[6]	vta	Vector Tail Agnostic policy. 1 = tail elements may be overwritten.
[5:3]	vsew	Selected element width: 000=SEW8, 001=SEW16, 010=SEW32.
[2:0]	vlmul	Register grouping multiplier: 000=LMUL1, 001=LMUL2.

Table 4.6: Bit-field map of vtype (0xC21, read-only).

vcsr Bit-Field Map

Table 4.7 gives the layout of vcsr (0x00F), which provides software a combined read/write view of vxrm and vxsat without requiring two separate CSR accesses.

Bits	Field	Description
[31:3]	(rsvd)	Reserved, reads as zero.
[2:1]	vxrm	Fixed-point rounding mode (mirrors the vxrm register).
[0]	vxsat	Fixed-point saturation flag (mirrors the vxsat register).

Table 4.7: Bit-field map of vcsr (0x00F, read/write).

Because vtype and vcsr are both combinational views over the same underlying state registers, vill/vma/vta/vsew/vlmul for vtype, and vxrm/vxsat for vcsr, a write through either aggregate register or its constituent fields observes the same state with no risk of divergence between the two views.

4.4.5 VL Computation Logic for vsetvl/vsetivli

When vset_we_i is asserted, the CSR unit performs an atomic update of vtype and vl in a single clock cycle. This update proceeds in two combinational stages that execute in the same cycle: a vtype validity check, followed by the three-case vl computation algorithm defined by the RVV specification [2].

vtype Validity Check

The 8-bit vset_vtype_i field forwarded by the decoder (Section 4.3.5) is decoded into its constituent vlmul and vsew sub-fields and checked for legality before any register is updated:

- **LMUL decode.** The vlmul[2:0] field is decoded to a numeric register-grouping multiplier. Only LMUL=1 (3'b000) and LMUL=2 (3'b001) are implemented by this design; every other encoding asserts vlmul_illegal.
- **SEW decode.** The vsew[5:3] field is decoded to an element width in bits. SEW=8 (3'b000), SEW=16 (3'b001), and SEW=32 (3'b010) are supported; every other encoding asserts vsew_illegal.

- **Overflow check.** Even given an individually legal `LMUL` and `SEW`, the combination is rejected if `LMUL * SEW > VLEN`, since no physical vector register can hold a register group of that width under the implemented `VLEN`.

If any of these three checks fails (`illegal_vtype = 1`), `vill` is set and every vector configuration field, `vlmul`, `vsew`, `vta`, `vma`, and `vl`, is zeroed, per the RVV specification's mandated behaviour for an illegal `vtype` request. Otherwise, the configuration fields are updated directly from `vset_vtype_i` and `vl` is set from the computation described below. This is the same combinational legality check that underlies the `e32m2 floor(vl/2)` erratum noted in Chapter 3 and analysed in Section 4.6: `SEW32/LMUL2` passes the overflow check (`LMUL * SEW = 64 < VLEN = 128`) and is therefore accepted as legal, but the controller's element-count derivation for this combination is where the erratum actually originates, not the CSR unit's validity check.

Three-Case `vl` Algorithm

Given a legal `vtype`, the new value of `vl` is computed from the `rs1_is_x0_i`/`rd_is_x0_i` flags and the application vector length `avl_i`, following the case structure summarised in Table 4.8.

`rd=x0`	`rs1=x0`	Case	`vl` result
Yes	Yes	A, Keep	Preserve the existing `vl`. `vtype` may still be updated.
No	Yes	B, Set Max	`vl = VLMAX`. Requests the maximum possible vector length.
,	No	C, AVL	`AVL=0` $\Rightarrow$ `vl=0`; `AVL` $\leq$ `VLMAX` $\Rightarrow$ `vl=AVL`; `VLMAX < AVL < 2×VLMAX` $\Rightarrow$ `vl=VLMAX`; `AVL` $\geq$ `2×VLMAX` $\Rightarrow$ `vl=VLMAX`.

Table 4.8: Three-case `vl` computation algorithm.

Case A (`rd=x0`, `rs1=x0`) corresponds to the `vl`-unchanged form of `vsetvli`, used by software to update `vtype` (for example, to change `SEW` mid-loop) without disturbing an already-established `vl`. Case B (`rd`$\neq$`x0`, `rs1=x0`) is the maximum-`vl` request form, in which software defers entirely to the implementation's `VLMAX` for the current `vtype`. Case C is the general form, in which `vl` is derived from an explicit application vector length, either the value of `rs1` (`vsetvli`) or a zero-extended 5-bit immediate (`vsetivli`); note that the saturating behaviour for `VLMAX < AVL < 2×VLMAX` and `AVL` $\geq$ `2×VLMAX` both resolve to `vl=VLMAX`, ensuring the computed `vl` is never assigned a value the hardware cannot subsequently execute. The `rs1_is_x0_i`/`rd_is_x0_i` flags consumed here are exactly the two flags the decoder extracts for this purpose, as described in Section 4.3.5; the CSR unit is the sole consumer of those flags, and the decoder itself performs no part of this computation.

4.4.6 Fixed-Point Saturation (`vxsat`) Handling

The `vxsat` flag follows a sticky-OR protocol common to fixed-point saturating arithmetic units: it is set whenever the ALU's fixed-point datapath asserts `vxsat_set_i`, and once set it can only be cleared by an explicit software write to `vxsat` or `vcsr` via a standard CSR instruction. The next-state logic implementing this protocol is

$$\texttt{vxsat_n} = \texttt{vxsat_q} \mid \texttt{vxsat_set_i}$$

so that a saturation event occurring on any element of any saturating instruction, across however many cycles that instruction's execution spans, is captured and held regardless of whether software polls vxsat on the very next cycle or only after several subsequent instructions. If a standard CSR write to CSR_VXSAT or CSR_VCSR (csr_we_int) is asserted in the same cycle that the ALU also signals vxsat_set_i, the explicit software-written value takes priority over the sticky-set value, giving software a deterministic way to clear the flag even in the presence of a concurrent saturation event. This sticky behaviour is the reason vxsat is exposed through vcsr[0] as well as its own dedicated address: software performing tight saturating-arithmetic loops can poll and clear the flag through a single vcsr access without needing to address vxsat and vxrm separately.

4.4.7 Reset Behaviour and Reset State Summary

On deassertion of the active-low asynchronous reset rst_n, every CSR state register is cleared to zero *except* vill, which is asserted to 0. This asymmetry is deliberate: clearing every other field to zero would otherwise leave the configuration state indistinguishable from a legitimately configured SEW8/LMUL1 context, allowing a vector instruction to execute against a configuration software never actually requested. Forcing vill=0 on reset instead places the core in a defined, unambiguous illegal-vtype state that persists until software performs an explicit vsetvli or vsetivli, at which point the validity check of Section 4.4.5 either clears vill (legal configuration) or leaves it set (illegal request, fields re-zeroed).
Table 4.9 summarises the reset value of every state register.

Register / Field	Reset value	Notes
vill	1'b0	Signals an illegal state until vsetvl executes.
vxsat	1'b0	No saturation pending.
vxrm	2'b00	Round-to-Nearest-Up (rnu).
vlmul	3'b000	LMUL = 1 (no grouping).
vsew	3'b000	SEW = 8-bit, but unused while vill=1.
vta	1'b0	Tail-undisturbed policy.
vma	1'b0	Mask-undisturbed policy.
vl	0	No active elements.
vstart	0	Start at element 0.

Table 4.9: Reset state of all vxt_csr_rvv state registers.

A related, narrower reset-like behaviour governs vstart during normal operation, not just at power-on: the combinational write logic defaults vstart_n to 0 every cycle, so vstart is automatically cleared on every cycle unless a standard CSR instruction explicitly targets it. This is intentional and matches the RVV specification's requirement [2] that vstart be cleared by ordinary vector instruction execution, leaving it non-zero only in the cycle immediately following a precise trap that an exception handler has explicitly written for restart purposes.

4.4.8 Timing Considerations

All state registers in vxt_csr_rvv, vill, vxsat, vxrm, vlmul, vsew, vta, vma, vl, and vstart, update synchronously on the rising edge of clk, with an asynchronous active-low reset overriding any synchronous update in progress. All remaining logic in the module, the CSR read

mux, the standard-CSR write-data computation, the vl computation algorithm, and the vtype validity check, is purely combinational and introduces no additional latency, so csr_rdata_o is valid in the same cycle that csr_addr_i is presented. This combinational read path is what allows the controller to read back a freshly written CSR value at RETIRE without an extra cycle of delay, consistent with the same-cycle write-then-read philosophy that governs the negative-edge-clocked vector register file (Section 4.5).

Because three independent paths, standard CSR instructions, vsetvl/vsetivli, and the ALU's vxsat sticky-set, can each request a write to the register bank, the next-state multiplexer shown in Figure 4.3 resolves simultaneous requests with a fixed priority order. The vset_we_i path takes priority over the standard-CSR-instruction path: if both are asserted in the same cycle, the vsetvl update wins. In practice this ordering is never actually contested, since the controller asserts csr_we_i and vset_we_i from mutually exclusive FSM states (a single instruction is either a Zicsr instruction or a vsetvli/vsetivli, never both), so the priority rule exists chiefly to give the module a well-defined, simulation-clean behaviour rather than to resolve a hazard that can occur in practice. The vxsat sticky-set path, by contrast, genuinely can coincide with a standard CSR write to vxsat/vcsr in the same cycle, the ALU may be retiring a saturating instruction in the same cycle a separate CSR instruction targets the saturation flag, and that case is resolved as described in Section 4.4.6, with the explicit software write taking priority over the autonomous sticky-set.

Because every combinational path in the module is short, a small priority encoder for the write-data mux, a handful of comparators for the vl/vtype legality checks, and a flat read-data multiplexer, vxt_csr_rvv is not expected to be a critical-path-limiting block in the design; the longer combinational cones identified in Section 4.3.7 (decoder) and analysed further in Section 4.6 (controller FSM) are the more likely determinants of VXT's achievable clock frequency.

4.5 Vector Register File (vxt_regfile)

The vector register file is the storage backbone of VXT: every arithmetic, load/store, configuration, and move instruction ultimately reads from or writes to it. This section describes its organisation, its port structure, and the write-timing considerations that arose during verification, including a bug uncovered in the whole-register-move (vmovnr) path.

4.5.1 Organisation

vxt_regfile implements the 32 architectural vector registers v0–v31 mandated by the RVV specification, each VLEN bits wide. For the present implementation, VLEN = 128, giving a total storage capacity of 32 × 128 = 4096 bits. The register file is modelled as a flat two-dimensional array, logic [VLEN-1:0] regfile_q[32], with no internal banking or replication: every register is a single physical storage location, and LMUL grouping is realised entirely *above* the register file, in the controller and decoder, rather than through any special hardware structure within vxt_regfile itself.

This decision has a direct consequence for how multi-register groups are handled. When an instruction operates under LMUL=2 (or, for widening destinations, an implicit register-group write), the register file is presented with two independent register addresses one cycle apart, the base register vd and the next register vd+1, rather than a single wide access spanning two physical entries. The controller (Section 4.6) is responsible for sequencing these two accesses correctly; vxt_regfile itself has no notion of LMUL and simply services whatever address it is given on each port. This keeps the register file's own logic completely independent of

vector configuration state, at the cost of pushing the entire grouping responsibility into the controller's FSM sequencing.

4.5.2 Read/Write Port Structure

The register file exposes three ports, summarised in Table 4.10: two independent read ports and a single write port.

Port	Type	Description
`vs1_addr_i` / `vs1_rdata_o`	Read	Combinational read port for the first source operand. Address is driven directly by the decoder's `vs1_addr_o` (Section 4.3.3); data is available in the same cycle.
`vs2_addr_i` / `vs2_rdata_o`	Read	Combinational read port for the second source operand, used for the `vs2` operand in all OPIVV/OPIVX/OPIVI/OPMVV/OPMVX forms.
`vd_addr_i` / `vd_wdata_i` / `vd_we_i`	Write	Single write port, qualified by a write-enable asserted by the controller during RETIRE (or during intermediate beats of a multi-cycle operation).

Table 4.10: Read/write port structure of `vxt_regfile`.

Both read ports are purely combinational: `vs1_rdata_o` and `vs2_rdata_o` reflect the contents of the addressed register on the same cycle the address is presented, with no registered latency. This mirrors the combinational read philosophy of the CSR unit (Section 4.4.8) and is what allows the ALU to consume both operands in the very same cycle the controller enters ALU_EXEC (Section 4.2.3), without an additional register-read pipeline stage.
The single write port is shared by every instruction class that produces a vector-register result: ordinary ALU writeback, widening/narrowing results, reduction results (written to element 0 of `vd`), vector loads, and the various VMV move forms (Section 4.3.6). Because there is only one write port, the controller must serialise any instruction whose semantics would otherwise require more than one write per cycle, this is precisely why an LMUL=2 instruction occupies two cycles of ALU_EXEC rather than one, as discussed in Section 4.2.3.
In addition to the standard read and write ports, the register file provides a dedicated output, `v0_o`, which continuously exposes the contents of architectural register `v0`. This signal is connected directly to the Vector ALU and is used as the vector mask input for masked instruction execution. Since `v0` is the architecturally-defined mask register in the RVV specification, providing a dedicated output avoids consuming one of the general-purpose read ports solely for mask retrieval and allows mask evaluation to occur in parallel with normal operand reads.

4.5.3 Write-Timing Considerations

Unlike every other clocked block in VXT, the register file's write port is clocked on the *negative* edge of `clk`, while both read ports remain purely combinational and therefore track the register contents continuously. This deliberate choice exists to satisfy the same-cycle read-after-write requirement of the non-pipelined execution model: because VXT never advances to a new

instruction until the current one has fully retired, a dependent instruction is guaranteed to issue no earlier than the cycle *after* a write, but the controller's own internal sequencing (in particular, the multi-beat ALU and reduction paths described in Section 4.6) sometimes needs to write an intermediate result on one half-cycle and read it back as an operand within the same full clock period. Negative-edge writeback gives the write a full half-cycle of margin ahead of the following positive-edge read, eliminating the need for an explicit bypass/forwarding network that a fully positive-edge-clocked register file would otherwise require.

This negative-edge write-timing scheme was also the direct source of a verification bug uncovered while exercising the whole-register-move instructions (`vmv1r.v` through `vmv8r.v`, collectively referred to as `vmovnr` in the testbench). The `vmovnr` datapath was originally written to assert its write-enable and write-data in the same controller state as ordinary single-register ALU writeback, on the implicit assumption that the timing margin established for the general case would automatically cover the multi-register move as well. In simulation, this produced an off-by-one-cycle write for the second (and any subsequent) register of a multi-register `vmovnr` group: the first register's write landed correctly on the negative edge of the expected cycle, but the second register's write, issued from a different point in the controller's group-iteration logic, was found to be racing the read used by the testbench's same-cycle checking logic, and intermittently failed self-checking under the comprehensive sweep described in the verification chapter. The root cause and the specific fix applied to `vxt_controller`'s group-iteration sequencing are detailed in Chapter 6 (Verification); the relevant point for this chapter is that the bug was a controller sequencing fault rather than a flaw in `vxt_regfile` itself, the register file's negative-edge write port behaved exactly as specified once supplied with correctly-timed write-enable and address signals for every beat of the group move.

4.6 Vector Controller / FSM (`vxt_controller`)

The vector controller is the sequencing authority of VXT. It is the only block in the design that simultaneously observes the full set of decoded instruction flags, the live vector configuration from the CSR unit, and the completion handshakes from the ALU, the memory subsystem, and the scalar core. On the basis of those observations it decides, on every rising clock edge, which block should be doing work, what register it should be reading or writing, and whether the scalar core is permitted to advance to its next instruction. Every other block in VXT, the decoder, the CSR unit, the register file, the ALU, the VLSU, is either combinational or self-contained; none of them self-sequences. All sequencing authority is centralised here.

4.6.1 FSM State Overview

The controller implements a five-state Moore FSM. The five states, their purposes, and the instruction classes that visit each one are summarised in Table 4.11. The canonical trajectory through any vector instruction is: accept the instruction in IDLE, perform whatever execution work the instruction requires across zero or more middle states, then spend exactly one cycle in RETIRE to hand the result back to the scalar pipeline.

State	Entered from	Purpose
ST_IDLE	Reset; RETIRE	Waits for a valid incoming instruction. When one arrives, latches its decoded fields and immediately begins beat 0 of execution in the same cycle.
ST_DECODE	IDLE	Continues register-group copy operations that require more than one beat; used exclusively by multi-register vmv and whole-register-move instructions.
ST_ALU_EXEC	IDLE	Handles any ALU work that could not be completed in the IDLE cycle: the second beat of a two-register-group operation, or the waiting period for a multi-cycle multiply result.
ST_MEM_EXEC	IDLE	Issues memory transactions to the VLSU and waits for each beat's OBI completion before advancing.
ST_RETIRE	Any state	Signals completion to the scalar core, commits any pending configuration or CSR updates, clears all internal iteration state, and returns unconditionally to IDLE.

Table 4.11: FSM states of `vxt_controller`, their entry points, and their roles.

The number of middle-state visits varies sharply by instruction class. Configuration instructions (`vsetvli`/`vsetivli`) and standard CSR instructions visit no middle state at all, completing in the minimum two-cycle IDLE→RETIRE path. Single-register-group ALU instructions whose result is available within the same clock cycle as the compute request also take just two cycles. Instructions with two-register-group operands, multi-cycle multiply pipelines, or memory access require one or more additional cycles in a middle state. RETIRE is invariably one cycle regardless of how many middle-state cycles preceded it.

4.6.2 State Diagram

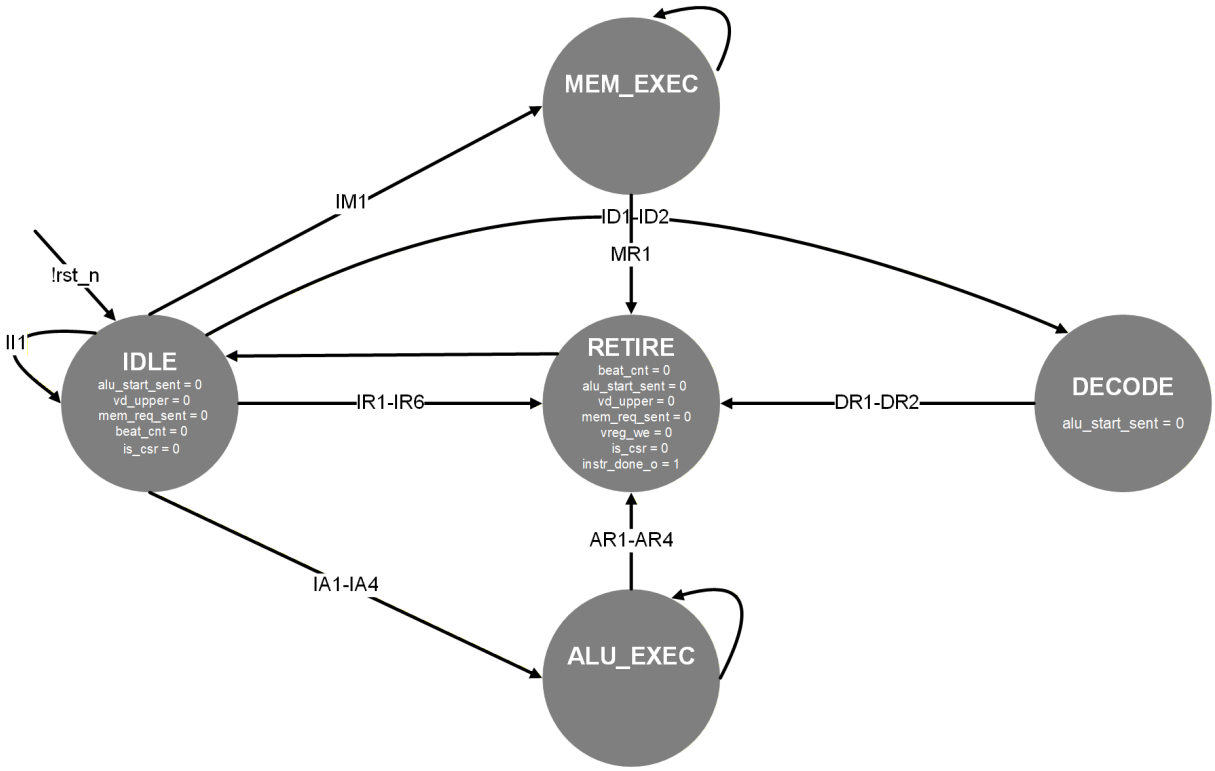

Figure 4.4: Complete FSM state-transition diagram for `vxt_controller`. Solid arcs are unconditional or always-taken transitions; labelled arcs carry the guard condition. The dashed self-loop on ST_MEM_EXEC represents the wait-for-beat-done stall. The dashed self-loop on ST_IDLE represents the scalar-bus-busy hold-off condition.

The principal arcs are: IDLE→RETIRE for all single-beat instructions (config, CSR, LMUL=1 ALU, LMUL=1 vmv, 1-register vmovnr);
IDLE→ALU_EXEC→RETIRE for LMUL=2 or multi-cycle ALU;
IDLE→DECODE→ $\cdots$ →RETIRE for multi-beat vmv/vmovnr;
IDLE→MEM_EXEC→RETIRE for LMUL=1 memory;
and IDLE→MEM_EXEC→MEM_EXEC→RETIRE for LMUL=2 memory.

The self-loop on IDLE fires only when a load or store arrives while the scalar core is using the shared OBI bus.

Table 4.12 provides a full legend of every guard condition label used on the arcs of Figure 4.4, grouped by the source state from which the arc departs.

Table 4.12: FSM transition condition legend for `vxt_controller`. Each label corresponds to an arc in Figure 4.4. Suffix convention: `I` = from IDLE, `A` = from ALU_EXEC, `M` = from MEM_EXEC, `D` = from DECODE; `R` = to RETIRE, `A`/`M`/`D` = to that state, `I` = self-loop. Registered signals end in `_r`; combinational inputs end in `_i`.

Label	Destination state	Guard condition (SystemVerilog)
From ST_IDLE		
IR1	RETIRE	`is_config_i`
IR2	RETIRE	`is_vmv_i`

Continuation of Table 4.12		
Label	**Destination state**	**Guard condition (SystemVerilog)**
IR3	RETIRE	`is_csr_i`
IR4	RETIRE	`!is_mul_i`
IR5	RETIRE	`is_vmovnr_i && vs1_i[2:0] == 3'd0`
IR6	RETIRE	`alu_valid_i && !is_widen_i`
IA1	ALU_EXEC	`!is_mul_i && lmul_eff_i == 2'd2`
IA2	ALU_EXEC	`alu_valid_i && !is_widen_i && lmul_eff_i == 2'd2`
IA3	ALU_EXEC	`alu_valid_i && is_widen_i`
IA4	ALU_EXEC	`is_mul_i`
ID1	DECODE	`is_vmovnr_i && !(vs1_i[2:0] == 3'd0)`
ID2	DECODE	`is_vmv_i && (lmul_eff_i == 2'd2)`
IM1	MEM_EXEC	`is_load_i || is_store_i`
II1	Self-loop (IDLE)	`is_load_i || is_store_i || data_req_core_i || scalar_rvalid_seen_r`
From ST_ALU_EXEC		
AR1	RETIRE	`!alu_start_sent_r && !is_mul_r && alu_valid_i && !(beat_cnt_r < (lmul_r - 2'd1))`
AR2	RETIRE	`!alu_start_sent_r && is_mul_r && alu_valid_i && !is_widen_r && !(beat_cnt_r < (lmul_r - 2'd1))`
AR3	RETIRE	`alu_start_sent_r && alu_valid_i && is_widen_r && (!alu_result_part_i) && !(beat_cnt_r < (lmul_r - 2'd1))`
AR4	RETIRE	`alu_start_sent_r && alu_valid_i && !is_widen_r && !(beat_cnt_r < (lmul_r - 2'd1))`
From ST_MEM_EXEC		
MR1	RETIRE	`mem_req_sent_r && mem_beat_done && !(beat_cnt_r < (lmul_r - 2'd1))`
From ST_DECODE		
DR1	RETIRE	`is_vmv_r && is_vmovnr_r && !(beat_cnt_r < vs1_r[2:0])`
DR2	RETIRE	`is_vmv_r && is_vmovnr_r && !(beat_cnt_r < (lmul_r - 2'd1))`

The five principal arc families visible in Figure 4.4 are as follows. The IDLE→RETIRE family covers every instruction class that completes its entire execution work within the IDLE accept cycle: configuration, CSR, single-register-group ALU, single-register-group vmv, and single-register whole-register moves. The IDLE→ALU_EXEC→RETIRE family covers two-register-group ALU operations and all multiply variants, with ALU_EXEC visited once per remaining register-group beat plus once per additional multiply-pipeline latency cycle. The IDLE→DECODE[→DECODE]* →RETIRE family covers vmv and whole-register-move instructions that need more than one register-group beat, with DECODE visited once for each beat beyond the first. The IDLE→MEM_EXEC[→MEM_EXEC]* →RETIRE family covers all loads and stores, with MEM_EXEC self-looping until each beat's OBI transaction completes. The IDLE self-loop fires only when a memory instruction arrives while the shared data bus is already occupied by a scalar core access, causing the controller to defer for one cycle and retry.

4.6.3 State-by-State Functional Description

`ST_IDLE`: Instruction Acceptance and Early Execution

IDLE is the resting state. The controller waits here until the scalar core presents a valid instruction. For all vector arithmetic and memory instructions, acceptance additionally requires that the active vector configuration held in the CSR unit is currently legal; an illegal configuration causes the controller to suppress acceptance and leave the instruction pending until software has issued a correcting `vsetvli`. Configuration instructions themselves (`vsetvli` and `vsetivli`) are the one class accepted unconditionally even in the presence of an illegal configuration, since they are the mechanism by which software establishes a legal one.
On the cycle an instruction is accepted, IDLE performs two things simultaneously. First, it records a stable copy of every decoded field, the destination and source register indices, the instruction class flags, the effective register-group multiplier, the element width, and for memory instructions the base address and stride, so that subsequent states have stable references to work from regardless of what the decoder presents in the following cycle. Second, it begins beat 0 of execution immediately, without waiting for a separate routing cycle. This early-start policy is the most consequential latency optimisation in the design: the original implementation spent one full cycle in a routing state before doing any useful work, and eliminating that cycle reduces the latency of every instruction class by one clock.
How beat 0 is executed depends on the instruction class:

- **ALU instructions (non-multiply).** The compute unit is enabled in the same cycle the instruction is accepted, reading the source registers directly from the as-yet-unregistered decoder outputs since no beat offset is needed for the first beat. When the result is combinationally available on that same cycle, which is the case for all integer arithmetic, shift, comparison, merge, and reduction instructions, the destination register write is committed immediately and the FSM proceeds directly to RETIRE. For two-register-group operations, only the first group's result is captured in IDLE; the FSM advances to ALU_EXEC to handle the second group.

- **Multiply instructions.** The multiply unit is enabled in IDLE exactly as for non-multiply ALU instructions. For narrow element widths (8-bit and 16-bit), the multiplier is fast enough that its result is available in the same cycle, and the controller treats it identically to a combinatorial ALU result. For 32-bit elements and for the upper-half-only multiply variants, the result is not available immediately; the controller records that the multiply has been started and advances to ALU_EXEC to wait for completion.

- **Memory instructions (loads and stores).** The VLSU is issued its first transaction request in the same IDLE cycle, provided the shared OBI data bus is not concurrently occupied by the scalar core. The controller marks internally that this first request has already been sent so that MEM_EXEC does not re-issue it on its first cycle. If the bus is occupied, the FSM self-loops in IDLE and retries on the next cycle. The beat-1 memory address for two-register-group memory operations is also pre-computed in IDLE using a shift rather than a multiply, avoiding area-intensive arithmetic in later states.

- **Register-copy moves (vmv).** The first destination register write is committed in IDLE itself. For single-register-group operations this is the only write, and the FSM proceeds directly to RETIRE. For two-register-group operations the beat counter is initialised to one and the FSM advances to DECODE to write the remaining register.

- **Whole-register moves (vmovnr).** The first register copy is performed in IDLE. Single-register moves retire immediately. Multi-register moves advance to DECODE, which iterates over each remaining register in the group.
- **Configuration and CSR instructions.** No register activity occurs in IDLE for these classes. The FSM routes directly to RETIRE, where a single-cycle pulse commits the configuration or CSR update.

`ST_DECODE`: Register-Copy Continuation

DECODE is entered exclusively by instructions that copy register content across more than one register-group beat: two-register-group vector-fill moves (`vmv.v.x`/`vmv.v.i`) and multi-register whole-register moves (`vmv`⟨`nr`⟩`r.v`). Beat 0 has already been written in IDLE; DECODE is responsible only for beats 1 through $N-1$.
For vector-fill moves, each cycle in DECODE writes one additional destination register, stepping sequentially through the destination register group. The source data (a scalar value or an immediate) is the same for every beat, so there is no source advancement. The state iterates until all registers in the group have been written, then transitions to RETIRE.
For whole-register moves, the behaviour is identical in structure, but the source address also advances by one register per beat, since the operation copies a contiguous block of source registers into a contiguous block of destination registers. The number of iterations is determined by the encoding of the move instruction itself, which specifies the group size directly.
It was in this state, and specifically in the timing of when each beat's register write is committed relative to the clock edge, that the write-racing bug described in Section 4.5.3 was rooted. The corrected implementation commits each beat's write on the entry cycle of the corresponding DECODE visit rather than on the following cycle, aligning all writes to the same half-edge margin that the register file's write port was designed to exploit.

`ST_ALU_EXEC`: Second-Beat and Multi-Cycle Completion

ALU_EXEC is entered when the execution work begun in IDLE is not yet complete. This occurs in two distinct situations.
The first is a two-register-group operation whose compute result is available combinationally. In this case ALU_EXEC is visited exactly once, on behalf of the second register group. The compute unit is re-enabled with source addresses offset to the upper half of each register group, and the result is written to the corresponding upper destination register in the same cycle. The FSM then advances to RETIRE.
The second situation is a multiply instruction whose result requires more than one pipeline cycle to produce. The controller waits in ALU_EXEC, holding all other activity, until the multiply unit signals that a result is ready. When the signal arrives, the behaviour diverges by instruction sub-type:

- **Non-widening multiplies** produce one result per register-group beat. Each result is written to the corresponding destination register. If more register-group beats remain, the multiply unit is re-enabled for the next beat and the wait resumes; otherwise the FSM advances to RETIRE.
- **Widening multiplies** produce a double-width result that must span two adjacent destination registers. The upper half of the widened result is produced first by the arithmetic pipeline and is written to the higher-addressed of the two destination registers; the lower half follows one cycle later and is written to the lower-addressed register. This two-cycle,

two-register write sequence occurs once per register-group beat, so a two-register-group widening multiply writes into four destination registers in total.

- **Widening multiply-accumulate** follows the same two-half write sequence as plain widening multiply, with one additional cycle per beat spent reading back the current contents of the destination register pair before the accumulation can proceed. The accumulated value for each half is therefore formed by combining the fresh partial product with the pre-existing destination value, requiring the old destination contents to be available as a third operand to the ALU.

`ST_MEM_EXEC`: Memory Transaction Sequencing

Mem_Exec manages the handshake between the controller and the vector load/store unit (VLSU) for each memory beat. Because the first transaction was already issued in Idle, Mem_Exec enters its waiting phase immediately on the first visit without re-issuing the request.
The completion criterion differs between unit-stride and strided accesses. For unit-stride operations, the controller considers a beat complete when the OBI bus delivers a valid read-data response. For strided operations, completion is signalled when the VLSU de-asserts its busy indication after the request has been sent.
When a beat completes:

- For a load, the data that the VLSU assembled from memory is written into the appropriate destination register of the register group.
- For a store, no register write occurs; the register file was read by the VLSU directly over the course of the transaction.
- If the operation covers more than one register-group beat, the controller clears the "request sent" marker and issues the second transaction, using a pre-computed address for the second beat's memory region. The state self-loops until that beat also completes.
- Once all beats are done, the FSM advances to Retire.

Throughout Mem_Exec, and during the initial transaction issue that occurs in Idle, the controller asserts an OBI bus ownership signal that grants the VLSU exclusive access to the shared data bus and prevents the scalar core from issuing new data-bus transactions until the vector memory operation is complete.

`ST_RETIRE`: Completion and Pipeline Release

Retire is always exactly one cycle long, regardless of how many cycles the preceding execution states required. Its sole function is to close out the current instruction and prepare for the next one.
On the Retire cycle, the controller asserts the instruction-done signal, which releases the scalar core from the stall it has been holding since the instruction was accepted. If the retiring instruction was a configuration instruction, a single-cycle pulse is sent to the CSR unit to commit the new vector type and vector length atomically. If it was a standard CSR instruction, a corresponding single-cycle pulse commits the read-modify-write operation on the targeted CSR. All internal iteration counters and state flags, beat position, multiply-started markers, memory-request-sent markers, and register-write-enable state, are cleared unconditionally, so the FSM is in a pristine condition when it returns to Idle on the following cycle.

One important distinction from a conventional pipeline: no vector register write is initiated *during* RETIRE. Every register write that the instruction produces is committed on the negative clock edge of whichever execution-state cycle last computed or received the corresponding data, in IDLE for single-beat instructions, in DECODE for the final beat of a multi-register move, in ALU_EXEC for the last ALU result, or in MEM_EXEC for the final memory beat. The RETIRE cycle therefore adds no register-file activity; its contribution is purely the done handshake and the configuration-commit pulses.

4.6.4 Instruction Execution Latency

Table 4.13 gives the total clock-cycle latency for every supported instruction class, measured from the cycle the instruction is accepted in IDLE through and including the RETIRE cycle. Four variable quantities determine latency for the non-fixed classes: R, the OBI round-trip latency per beat for unit-stride memory operations; S, the VLSU busy duration per beat for strided operations; K, the multiply pipeline depth in cycles; and NR, the number of registers in a whole-register-move group. Every latency expression includes one cycle for IDLE and one for RETIRE, so the minimum for any instruction is two cycles.

Instruction class	**FSM trajectory**	**LMUL = 1**	**LMUL = 2**
Configuration			
`vsetvli` / `vsetivli`	IDLE→RETIRE	2	2
Standard CSR (Zicsr)			
`csrrw` / `csrrs` / `csrrc`	IDLE→RETIRE	2	2
Unit-stride load / store			
`vle.v` / `vse.v`	IDLE→MEM→RETIRE	$R+2$	$2R+4$
Strided load / store			
`vlse.v` / `vsse.v`	IDLE→MEM→RETIRE	$S+2$	$2S+4$
Vector fill and scalar-element move			
`vmv.v.x` / `vmv.v.i`	IDLE [→DEC] →RETIRE	2	3
`vmv.s.x` / `vmv.x.s`	IDLE→RETIRE	2	2
Whole-register move			
`vmv⟨nr⟩r.v`	IDLE→DEC^{NR-1} →RETIRE	$NR+1$	$NR+1$
Combinatorial ALU			
All combinatorial ALU	IDLE [→ALU] →RETIRE	2	3
Non-widening multiply			
`vmul` / `vmulh`	IDLE→ALU^{K-1} →RETIRE	$K+2$	$2K+3$
Widening multiply			
`vwmul` / `vwmulu`	IDLE→ALU^{K} →RETIRE	$K+3$	$2K+5$
Widening multiply-accumulate			
`vwmacc`	IDLE→ALU^{K+1} →RETIRE	$K+4$	$2K+7$

Table 4.13: Execution latency by instruction class. R: OBI round-trip per memory beat. S: VLSU busy duration per strided beat. K: multiply pipeline depth (1 for SEW = 8/16 fast path; 2 for SEW = 32; 3 for `vmulh` SEW = 32). NR: number of registers in a whole-register-move group (1–8). Square brackets indicate a state visited only for LMUL = 2.

The dominant observation is that the early-start optimisation reduces the baseline latency of almost every instruction class by one cycle relative to the original implementation. As a result, the practical latency for the most common operations, single-register-group integer arithmetic, comparisons, shifts, and vector fill, is two cycles, which is the theoretical minimum achievable by any non-pipelined implementation that must both accept an instruction and signal its completion to the scalar core. Two-register-group variants of the same operations add exactly one additional cycle. Multiply latency is determined entirely by the multiply pipeline depth and is independent of the FSM structure.

4.6.5 Special Write-Enable Paths and Whole-Register-Move Handling

The controller is responsible for committing results to three distinct destinations, each with its own timing and enabling rules.

Vector register writes. Register-file writes are the most timing-critical commitment in the design because they land on the negative clock edge and must therefore be enabled on the correct positive-edge-to-positive-edge cycle with no ambiguity. The write-enable and write-address are set up by whichever execution state last produced or received a result, so that the write lands on the negative edge of the transition out of that state. The one override to this rule concerns the `vmv.x.s` instruction, which extracts a single element from a vector register and delivers it to the scalar integer register file rather than writing back to the vector register file. For this instruction the vector register write-enable is suppressed entirely, even though the controller's general move-detection logic would otherwise assert it.

Configuration commits. When a `vsetvli` or `vsetivli` instruction retires, the controller asserts a one-cycle commit pulse to the CSR unit. This pulse is what triggers the atomic update of the vector type and vector length, as described in Section 4.4.5. The pulse is active for exactly one cycle during RETIRE and is never asserted in any other state.

Standard CSR commits. When a Zicsr instruction retires, a separate one-cycle commit pulse is asserted to the CSR unit, carrying with it the already-decoded operation type and write-data value that have been stable in registered form since IDLE. This pulse and the configuration commit pulse are mutually exclusive within any single RETIRE cycle: an instruction is classified as either a configuration instruction or a Zicsr instruction in IDLE, never as both.

Whole-register-move write sequencing. The whole-register-move instructions require particular care because they produce one register write per cycle across multiple consecutive DECODE visits, and the register file's negative-edge write timing means the enable for each write must be asserted on exactly the right cycle. The corrected implementation, described in detail in Section 4.5.3, asserts the write-enable during the DECODE visit that corresponds to that beat, so the write lands on the negative edge of that same cycle. The enable is cleared automatically as the FSM transitions, ensuring no spurious write occurs during the subsequent RETIRE cycle. The first beat's write is handled in IDLE, consistent with the early-start policy, and is indistinguishable in timing from the writes produced by any other single-beat instruction.

4.6.6 Widening Operations and Reduction Handling

Widening Arithmetic

A widening instruction multiplies two operands of a given element width and produces a result at twice that width. Because the vector register file stores registers at the native element width, the double-width result must be split across two adjacent destination registers, the lower half of the product into the lower-addressed register and the upper half into the register immediately above it.
The arithmetic unit handles this split internally, producing the upper half first and the lower half one cycle later. The controller responds to these two result tokens in the order they arrive, committing each to the appropriate destination register in the same cycle it becomes

available. For a single-register-group widening operation this occupies two consecutive cycles of ALU_Exec; for a two-register-group widening operation the same two-cycle sequence is repeated for the second group, occupying four cycles of ALU_Exec in total and writing into four consecutive destination registers.
Widening multiply-accumulate extends this sequence by one additional cycle per register-group beat. Before computing the product and adding it to the existing destination value, the controller must supply the current contents of the destination register pair to the arithmetic unit as a third operand. One cycle of each beat is therefore spent reading those contents from the register file and presenting them to the accumulate path, after which the two-cycle upper-then-lower result write proceeds as for plain widening multiply.

Narrowing Operations

Narrowing operations are the logical inverse: they take a double-width source operand (which spans two adjacent source registers) and produce a single-width result that fits in one destination register. The controller handles this by treating narrowing instructions as inherently two-beat regardless of the configured register-group multiplier: two source register reads are needed to cover both halves of the double-width input, and both contribute to a result that is packed into the single destination register. The first beat produces the lower portion of the packed result and the second beat produces the upper portion; both writes go to the same destination register address with no offset between them.

Reduction Operations

Reduction instructions accumulate all active elements of a source register group into a single scalar result, which is deposited in element 0 of the destination register. For a single-register-group reduction the accumulation is performed in one ALU pass and the result is written immediately, giving a two-cycle total latency. For a two-register-group reduction the accumulation must span both groups. The controller instructs the arithmetic unit to retain the intermediate accumulation result from the first group internally rather than writing it out; in the second group's beat the arithmetic unit combines that saved intermediate with the second group's elements to produce the final scalar result. Both beats write to the same destination register address (element 0 of the destination) since the final result is always a single element regardless of how many register-group beats were required to compute it.

4.6.7 VTA/VMA Policy Enforcement at the Controller Level

The RVV specification [2] defines two agnostic policies governing what happens to elements that are not updated by a given instruction. The *tail agnostic* policy (VTA) applies to elements beyond the current vector length; the *mask agnostic* policy (VMA) applies to elements whose corresponding mask bit is inactive. Under the "agnostic" setting for either policy, the implementation may write any value to those elements; under the "undisturbed" setting, the elements must retain exactly the value they held before the instruction executed. Software selects between agnostic and undisturbed independently for tails and mask-off elements by writing the appropriate bits in the vector type CSR.
VXT divides the responsibility for enforcing these policies cleanly between the controller and the arithmetic unit. The arithmetic unit is responsible for applying the policy element-by-element within each beat: it tracks which element positions are active, which are tail, and which are mask-off, and it selects the appropriate source, either the computed result, the preserved old value, or an arbitrary fill, for each position in the writeback. The controller is

responsible for ensuring that the arithmetic unit always has the information it needs to make those per-element decisions. This controller-level contribution takes three forms.
First, the current policy settings are sourced directly from the live CSR outputs on every cycle, without being registered in the controller. This ensures that the policy bits presented to the arithmetic unit always reflect the most recently committed configuration, with no possibility of executing a beat against a stale setting left over from a prior `vsetvli`.
Second, the controller informs the arithmetic unit which register-group beat is currently being processed. This information, combined with the current vector length from the CSR unit, allows the arithmetic unit to determine precisely which element positions within the current beat's VLEN-wide data word are active and which are tail. A beat that covers elements entirely within the active length has no tail elements to handle; a beat that extends beyond the active length may have a mix; a beat that lies entirely beyond the active length consists entirely of tail elements.
Third, the register file provides a dedicated, continuously-driven output exposing the contents of the architectural mask register at all times. This mask value reaches the arithmetic unit without being routed through the controller at all: the controller neither arbitrates nor qualifies it. The arithmetic unit selects the appropriate bit from the mask register for each element position using the current beat index and position, and uses it to determine whether each element is mask-active or mask-off in the VMA sense.
The consequence of this partitioning is that the controller adds zero FSM states to handle VTA or VMA policy. Because the old destination register contents are available on a dedicated read port of the register file with zero latency, the undisturbed-policy case requires no stall or prefetch cycle: the arithmetic unit can read the old value and select it combinationally for any element that the policy requires to be preserved. The agnostic-policy case is even simpler, requiring no old-value read at all. In both cases the same two-cycle minimum latency applies to any single-register-group instruction, regardless of which policy combination is in effect.

4.7 Vector Load/Store Unit (`vxt_vlsu`)

The Vector Load/Store Unit (`vxt_vlsu`) is the memory subsystem of VXT. It translates each `vle.v`, `vse.v`, or `vlse.v` instruction into a sequence of OBI bus transactions, enforces the full set of RVV element-level execution policies (`vstart`, `vta`, `vma`, and masking), and presents a single-cycle completion signal to the controller FSM once every active element has been transferred. Unlike the ALU, which is entirely combinational within a beat, the VLSU contains its own internal sequencer FSM that manages the OBI handshake independently of the outer controller, allowing the outer FSM to remain in a simple wait state while the VLSU handles the complexities of bus arbitration, pipelined grant/response decoupling, and strided window batching.

4.7.1 Overview and OBI-Based Transaction Model

Two properties shape every block in the VLSU:

1. **One 128-bit OBI word per unit-stride beat.** With VLEN = 128, a single unit-stride load or store transfers exactly one full vector register per OBI transaction. The byte-enable mask (`data_be_o`) restricts which of the 16 bytes carry live data, so no data reassembly is ever needed for unit-stride accesses.

2. **Pipelined grant/response decoupling.** The OBI protocol permits the grant and the read-data response to arrive in different cycles. The VLSU tracks these two events on

independent counters (`req_cnt_q` for request-side progress, `pack_cnt_q` for response-side accumulation), allowing outstanding transactions to be in-flight without stalling the request pipeline.

The module supports three access modes: unit-stride load (`vle.v`), unit-stride store (`vse.v`), and strided load (`vlse.v`). Strided store is intentionally omitted in this revision; the rationale is documented in Section 4.7.5. All three modes share the same OBI address and data buses, the same byte-enable generation logic, and the same element-active predication infrastructure. The distinguishing factor is how the address advances across transactions (base fixed versus base + stride × window index), and how returning read data is assembled into the destination register.

4.7.2 Block Diagram

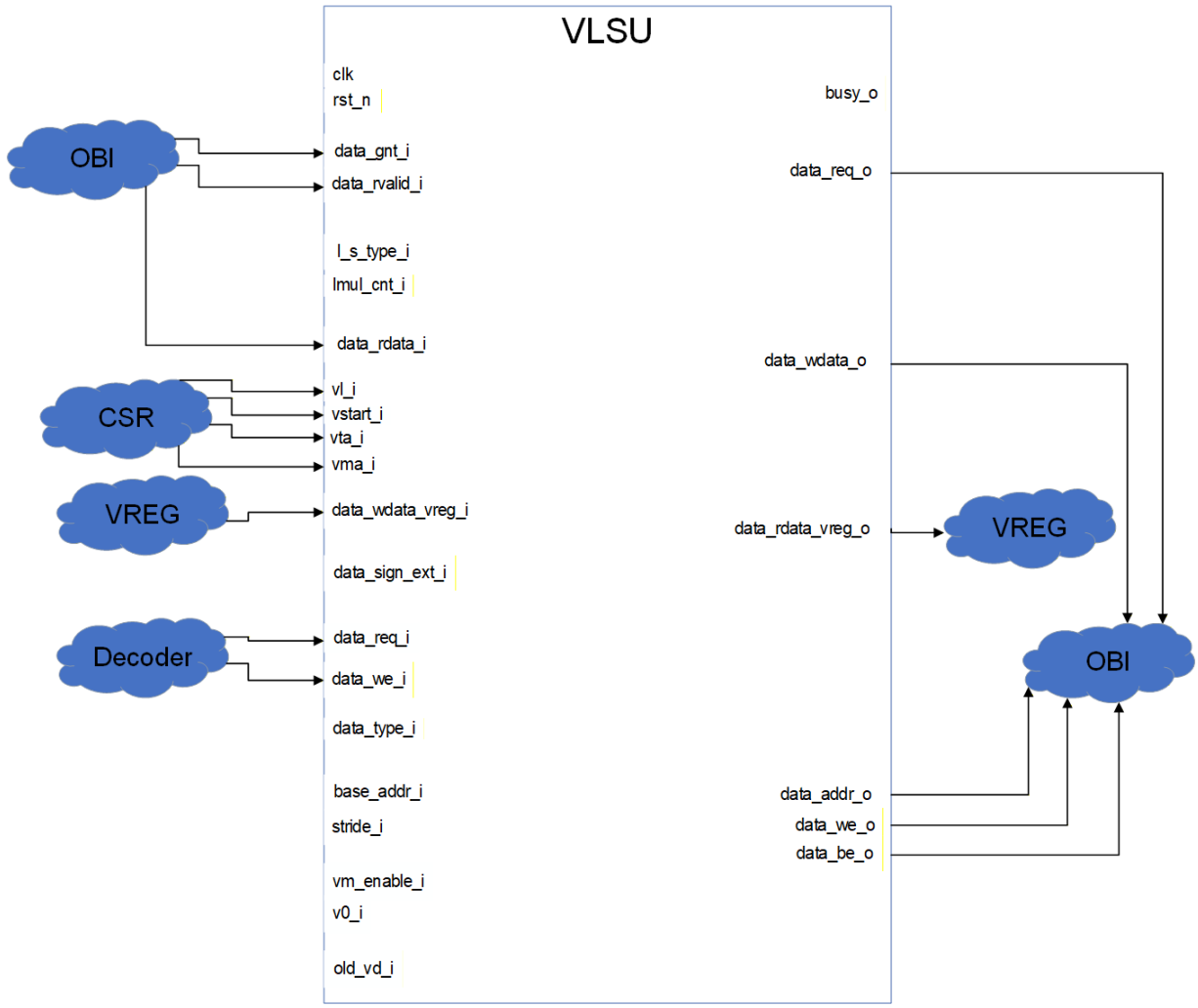

Figure 4.5: Internal microarchitecture of `vxt_vlsu`. Eight functional blocks collaborate under the direction of the internal Sequencer FSM. Data flows from the OBI response bus through the Unit-Stride Load Merge logic or the Strided Logic block into the register file write port. Control flows from the Effective Parameters block through the Byte Enable and Address Generation logic to the OBI request port. Dashed borders indicate combinational blocks; solid borders indicate blocks that contain registers.

The module is internally organised into eight functional blocks, each with a well-defined responsibility boundary:

- **Effective Parameters Calculation.** Converts raw CSR values into beat-local parameters used by every downstream block: `sew_enc`, `bytes_per_elem`, `elems_per_beat`, `beat_offset`, `eff_vl`, `eff_vstart`, and the per-element active flag array `elem_active_w`.
- **Byte Enable Logic.** Produces the 16-bit `trans_be` that determines which bytes of the 128-bit OBI bus word carry valid data for each transaction.
- **Address Generation and Transaction Valid Logic.** Computes `trans_addr` for each OBI transaction and controls when `trans_valid` is asserted.
- **Store Path.** A pass-through that routes `data_wdata_vreg_i` to `trans_wdata` and asserts `trans_we`; byte-enable restrictions limit which bytes actually reach memory.
- **Strided Logic.** For strided loads, unpacks each OBI response window into the `rdata_q` accumulator register, applying `vma` policy per element.
- **Unit-Stride Load Merge Logic.** Merges raw OBI response data with `old_vd_i` according to the four-way element policy (prestart, active-masked, active-unmasked, tail) for unit-stride loads.
- **Sequencer FSM.** The central state machine that drives all transaction sequencing, manages all registered state, and produces `data_rdata_vreg_o`.
- **OBI Interface Adapter.** Adapts internal transaction signals to the standard OBI handshake with `TRANS_STABLE=1` semantics.

4.7.3 Byte-Enable / Active-Element Logic (`elem_active_w`, Masking, Prestart/Tail Handling)

Every element slot in the 128-bit bus word must be individually classified before any transaction can be issued. The Effective Parameters block generates a combinational flag vector `elem_active_w[VLMAX-1:0]` (up to 16 flags, one per element slot) by evaluating three simultaneous conditions for each element index i within the current beat:

$$\texttt{elem_active_w}[i] = \underbrace{(i_{\text{global}} \geq \texttt{eff_vstart})}_{\text{not prestart}} \wedge \underbrace{(i_{\text{global}} < \texttt{eff_vl})}_{\text{not tail}} \wedge \underbrace{(\neg\, \texttt{vm_enable_i} \vee \texttt{v0_i}[\texttt{global_elem_base} + i])}_{\text{not masked-off}}, \quad (4.1)$$

where $i_{\text{global}} = \texttt{global_elem_base} + i$ is the element's global index across the LMUL group, `eff_vstart` and `eff_vl` are the beat-local effective start and length computed by the Effective Parameters block, and `global_elem_base` offsets into the mask register `v0` correctly per LMUL beat (beat 0 maps to `v0[0..elems_per_beat-1]`; beat 1 maps to `v0[elems_per_beat]` through `v0[2 · elems_per_beat − 1]`).
This single flag vector then drives both the Byte Enable Logic and the Unit-Stride Load Merge Logic, so prestart, tail, and mask-off exclusions are applied uniformly in one place rather than being independently re-derived in multiple blocks.

Unit-stride byte enable. For unit-stride accesses, each OBI bus byte k belongs to a specific element whose index is a synthesis-time constant:

$$\texttt{ei}(k) = \lfloor k/\texttt{bytes_per_elem} \rfloor.$$

Byte k is enabled if and only if `elem_active_w[ei(`k`)]` is asserted:

$$\texttt{trans_be_unit}[k] = \texttt{elem_active_w}[\lfloor k/\texttt{bytes_per_elem} \rfloor].$$

Because the condition is computed at elaboration time (constant element-to-byte mapping), the resulting byte-enable hardware is a bank of simple wires, no runtime arithmetic is required.

Strided byte enable. For strided window loads, element j within a window begins at byte offset $j \times$ `stride_i` within the 128-bit bus word. Each byte b within that element contributes bit $j \cdot$ `stride_i` $+ b$ to `trans_be`. All $16 \times 4 = 64$ contributions (16 element slots, up to 4 bytes each) are OR-reduced through a synthesis-friendly binary tree to produce `trans_be_strided_w`. A final multiplexer selects between the unit-stride and strided byte-enable paths based on `l_s_type_i`.

Policy application at load merge time. Byte enables govern *which bytes are presented to or from the OBI bus*, but they do not by themselves implement the undisturbed portions of the `vta` and `vma` policies, since those require selecting between the arriving memory data and the preserved `old_vd_i` value. That selection is performed in the Unit-Stride Load Merge Logic (Section 4.7.7) and the Strided Logic (Section 4.7.6).

4.7.4 Address Generation and Transaction-Valid Logic

The address presented on `data_addr_o` for each OBI transaction is determined by the access mode.

Unit-stride. The address is fixed at `base_addr_i` for the entire beat. Because all 16 bytes are part of a single naturally-aligned 128-bit transfer, no per-element address arithmetic is needed; the byte-enable mask provides all element selectivity.

Strided. The address of the n-th request window is

$$\texttt{trans_addr} = \texttt{base_addr_i} + \texttt{req_cnt_q} \times \texttt{stride_i},$$

where `req_cnt_q` is the first element index in the current request window. The VLSU asserts `trans_valid` while the sequencer is busy and `req_cnt_q` $<$ `eff_vl`. After all request windows are issued (`req_cnt_q` $\geq$ `eff_vl`), `trans_valid` falls and OBI requests stop.

OBI stability requirement (`TRANS_STABLE=1`). The OBI protocol requires that address and control signals remain stable from the first assertion of `data_req_o` until `data_gnt_i` is received. The VLSU satisfies this naturally: for unit-stride `trans_addr` is a combinational function of the constant `base_addr_i`; for strided, `req_cnt_q` does not advance until after `data_gnt_i` is asserted, so the address is held stable throughout each grant-wait period.

4.7.5 Store Path (Pass-Through, Byte-Enable Restriction)

The store path is simple by design: `data_wdata_vreg_i` (the full 128-bit contents of the source vector register read by the controller before asserting `vlsu_req_o`) is connected directly to `trans_wdata` with no reassembly, rotation, or element packing. `trans_we` is asserted when `data_we_i` is high. The byte-enable mask `trans_be`, generated by the Byte Enable Logic from `elem_active_w` , restricts which bytes of the bus word are actually committed to memory. Prestart bytes, tail bytes, and mask-off bytes are silently suppressed at the memory interface without requiring the register-file data presented on the bus to be modified.
Strided store is intentionally not supported in this implementation. A correct strided store would require a per-element address computation (one OBI write transaction per element or per window), a data-rotation mux to place each element at byte lane 0 of its own transaction, and a separate byte-enable computation per transaction. Eliminating this use case removes all per-element byte-lane rotation hardware from the store path, reducing the datapath to a single wire, and reduces OBI transaction count modelling for stores to a single 128-bit transfer per beat.

4.7.6 Strided Load Logic and Window-Batching Optimisation

Without optimisation, a strided load of N active elements would require N separate OBI transactions, each transferring a single element. When the stride is smaller than the OBI bus width, multiple elements fit within a single 128-bit response word, and batching them into one transaction reduces bus traffic and latency.

Window-size calculation (`elems_per_window_w`). Batching activates when all four conditions are satisfied: (1) the stride is positive (`stride_i[XLEN-1]=0`), (2) it is non-zero, (3) it is less than 16 bytes (fits within one bus word), and (4) it is at least `bytes_per_elem` (ensuring no byte-level overlap between adjacent elements within the same window). When batching is active, the number of elements per window is $\lfloor 16/\texttt{stride_i} \rfloor$. Table 4.14 lists the resulting window sizes for common stride values.

Stride (bytes)	`elems_per_window`
1	16
2	8
3	5
4	4
5	3
6	2
7	2
8	2
9–15	1 (no batching gain)
≥ 16	1 (no batching gain)

Table 4.14: Strided window-batching OBI transaction count as a function of stride for SEW=8 (`bytes_per_elem=1`). For SEW=16 or SEW=32 the effective window size is $\lfloor 16/\max(\texttt{stride_i}, \texttt{bytes_per_elem}) \rfloor$, so strides below `bytes_per_elem` disable batching entirely.

Element unpacking. For each arriving OBI response word, a fully-unrolled generate loop (over element indices $j = 0 \ldots \texttt{VLMAX} - 1$) extracts element j from its byte position within the 128-bit word and writes it into `rdata_q` at the correct global element offset. Each extracted element is subject to the `vma` policy: if `vm_enable_i` is asserted and `v0_i[global_elem_base + j]` is zero (masked off), the element slot in `rdata_q` takes `old_vd_i` (undisturbed) or `0xFF...F` (agnostic), according to `vma_i`.

Strided-done detection. The strided load is complete when the response window containing global element index `eff_vl` − 1 has been received on the pack side (`pack_cnt_q + elems_per_window_w` ≥ `eff_vl`). At that point `strided_load_done_w` fires, the sequencer deasserts `busy_q`, and the vta tail policy is applied combinationally across all `rdata_q` slots beyond `eff_vl` before the result is captured into `data_rdata_vreg_o`.

4.7.7 Unit-Stride Load Merge Logic (4-Way Priority Mux)

For unit-stride loads, the arriving 128-bit OBI response word cannot be written directly to the register file: inactive element slots (prestart, masked, tail) must retain their existing values according to the `vta` and `vma` policies, and the policy-selected values may differ from what memory returned. The Unit-Stride Load Merge Logic resolves this with a fully-combinational 4-way priority multiplexer applied independently to each bus byte i ($0 \leq i < 16$):

1. **Prestart** ($i_{\text{global}} <$ `eff_vstart`): output byte i takes `old_vd_i`[i]. Prestart elements are always undisturbed, regardless of `vta`.
2. **Active, masked-off** ($i_{\text{global}} \geq$ `eff_vstart`, $i_{\text{global}} <$ `eff_vl`, `vm_enable_i`=1, `v0_i`[`global_elem_base` + ei]=0): output byte i takes `old_vd_i`[i] if `vma_i`=0, or `0xFF` if `vma_i`=1.
3. **Active, unmasked** (all other body elements): output byte i takes `resp_rdata`[i], the byte returned by memory.
4. **Tail** ($i_{\text{global}} \geq$ `eff_vl`): output byte i takes `old_vd_i`[i] if `vta_i`=0, or `0xFF` if `vta_i`=1.

The priorities are checked in the order listed above; prestart takes precedence over tail in the degenerate case where `eff_vl` < `eff_vstart` (a legal but programmer-error condition). The logic is a generate block that synthesises 16 independent 4-input muxes, one per bus byte. It is fully combinational: `unit_load_merge_w` is valid in the same cycle that `data_rvalid_i` is asserted, and the Sequencer latches it directly into `data_rdata_vreg_o` once the response arrives. No extra pipeline stage is required between the OBI response and the register file.

4.7.8 Sequencer FSM (`busy_q`, `req_cnt_q`, `pack_cnt_q`, `rdata_q`) and OBI Handshake Timing

The Sequencer is the only stateful block in the VLSU. It holds six registers:

Register	Width	Description
busy_q	1	High while a strided sequence is in progress; deasserted one cycle after strided_load_done_w fires.
req_cnt_q	6	First element index of the *next request window*; advances by elems_per_window_w on each data_gnt_i.
pack_cnt_q	6	First element index of the *next response window*; advances by elems_per_window_w on each resp_valid.
rdata_q	128	Strided load accumulator; seeded with old_vd_i on data_req_i so prestart slots are already correct.
trans_done_q	1	One-cycle pulse flagging that the strided result is fully accumulated.
data_rdata_vreg_o	128	Final load result output to the register file; latched from unit_load_merge_w (unit-stride) or strided_final_w (strided).

Table 4.15: Sequencer FSM registers in vxt_vlsu.

OBI handshake timing. A minimum-latency unit-stride transaction completes in two cycles: cycle 0 asserts data_req_o and receives data_gnt_i simultaneously; cycle 1 receives data_rvalid_i together with data_rdata_i. The negative-edge latch on data_rdata_vreg_o captures the merged result at the end of cycle 1, making it available to the register file write port at the start of cycle 2.
For strided accesses, req_cnt_q and pack_cnt_q are decoupled: the request side advances on each data_gnt_i (which may occur in the same cycle as the request, or one or more cycles later under back-pressure), while the response side advances on each resp_valid (which arrives at least one cycle after the corresponding grant). This decoupling is essential for pipelined OBI operation: if the memory grants a new request while the previous response has not yet arrived, both counters may be in flight simultaneously. The VLSU is correct in this scenario because rdata_q element slots are written by index (pack_cnt_q + j) rather than by FIFO position, so responses may arrive and be packed in any order without corrupting the accumulator.

Unit-stride busy signal. For unit-stride accesses, busy_q is not used; instead the outer controller detects beat completion via the registered data_rvalid_i signal (Section 4.6.3). busy_o is therefore only meaningful for strided operations, where its deassertion signals the controller that one VLSU beat is complete.

4.7.9 LMUL=2 Multi-Beat Operation at the VLSU Level

For LMUL=2, the outer controller issues two back-to-back VLSU requests, distinguished by lmul_cnt_i: 0 for the first 128-bit half of the double-wide register group and 1 for the second. From the VLSU's perspective, each request is handled identically; the only LMUL-sensitive computation is in the Effective Parameters block.

Beat-local re-indexing. For LMUL=2, the total vector length may reach $2 \times$ `elems_per_beat` elements. Beat 0 processes global elements $[0, \texttt{elems_per_beat})$ and beat 1 processes elements $[\texttt{elems_per_beat}, 2 \times \texttt{elems_per_beat})$. The Effective Parameters block computes:

$$\begin{aligned} \texttt{global_elem_base} &= \texttt{lmul_cnt_i} \times \texttt{elems_per_beat}, \\ \texttt{eff_vl} &= \min(\texttt{vl_i} - \texttt{beat_offset},\ \texttt{elems_per_beat}), \\ \texttt{eff_vstart} &= \max(\texttt{vstart_i} - \texttt{beat_offset},\ 0), \end{aligned}$$

where $\texttt{beat_offset} = \texttt{lmul_cnt_i} \times \texttt{elems_per_beat}$. This re-indexing ensures that every downstream block (Byte Enable Logic, Address Generation, Strided Logic, Unit-Stride Load Merge Logic) always operates on 0-indexed element counts in the range $[0, \texttt{elems_per_beat}]$, regardless of which LMUL beat is being processed. If `vl_i` is smaller than `beat_offset`, `eff_vl` collapses to zero, all `elem_active_w` bits are deasserted, and the VLSU issues a no-op transaction (all bytes disabled), keeping the register-file contents undisturbed for the second half of the register group. This is the correct `vta=0` (undisturbed tail) behaviour for instructions where `vl` does not reach the second register.

Controller interaction. The outer controller (Section 4.6.3) handles LMUL=2 memory instructions by iterating ST_MEM_EXEC twice. The VLSU base address for beat 0 is `base_addr_i` (the instruction's `rs1` value). The beat 1 address is pre-computed in ST_IDLE as `base_addr_i` + `VLEN/8` (unit-stride) or `base_addr_i` + `stride_i` $\times$ `elems_per_beat` (strided), and is passed to the VLSU as `vlsu_base_addr_o` on the second `vlsu_req_o` assertion. This pre-computation eliminates a multiplier from the ST_MEM_EXEC critical path (Section 4.6.3).

4.8 Vector ALU (`vxt_alu`)

The vector arithmetic logic unit is the compute core of VXT. It executes all element-wise operations on vector register data, from simple integer addition to signed saturating arithmetic, mask generation, and multi-cycle widening multiplication, and reports its status back to the controller FSM via a three-signal handshake. The ALU is structured as a hierarchy of four cooperating sub-modules, each with a single, well-defined responsibility, rather than as a monolithic datapath. This separation keeps each sub-module individually verifiable and limits the combinational cone that any single synthesis optimisation pass must explore.
The four sub-modules and their role within the hierarchy are:

- `op_b_mux`: selects and broadcasts the correct second operand to every element lane before computation begins.
- `vxt_alu` **(top-level)**: performs all single-cycle element-wise compute operations, applies element-level policy, and produces the final `result_o` bus.
- `vxt_mul`: a separate, pipelined multiplier that handles the four multi-cycle multiply variants and signals completion via the same handshake the ALU top exposes to the controller.
- `vxt_reduction_tree`: accumulates a scalar result across all active elements of the source vector, supporting both sum and signed-maximum reductions with a registered partial-result path for LMUL=2.

The ALU top-level module integrates all four sub-modules and presents a single, unified interface to the rest of VXT. The controller (Section 4.6) drives the ALU via `enable_i` and reads back completion status on `valid_o`, `busy_o`, and `result_part_o`. No other block in VXT observes the ALU's internal sub-module boundaries.

4.8.1 Overview

Purpose and scope. The ALU performs every compute operation that does not require a memory transaction. This encompasses eight distinct operation classes: integer arithmetic (add/subtract/min/max), signed saturating arithmetic, logical and shift operations, widening and narrowing operations (double-width products, half-width shift-rights), mask-output comparisons, merge and conditional-select operations, element-index generation (`vid.v`), and the two supported reduction operations (`vredsum.vs`, `vredmax.vs`). Multiply-class operations (`vmul`, `vmulh`, `vwmul`, `vwmacc`) are forwarded internally to `vxt_mul` and return via the same `result_o` bus.

Controller handshake. The ALU receives a one-cycle `enable_i` pulse from the controller at the start of each ALU beat. For all single-cycle operations the ALU asserts `valid_o` in the same cycle as `enable_i`, allowing the controller to write back the result and advance to the next state without any wait. For multi-cycle multiply operations, `busy_o` is asserted while the pipeline is in progress and `valid_o` is deferred until the result is ready. The controller is required by protocol to refrain from asserting a new `enable_i` while `busy_o` is high; a simulation-only assertion in the RTL flags any violation of this contract.
The `result_part_o` signal is meaningful only for widening multiply and distinguishes the two sub-results that a widening operation produces from a single source pair: `1` identifies the upper VLEN-wide half (written first to `vd+1`) and `0` the lower half (written to `vd`). The controller uses this signal to route each sub-result to the correct destination register address, as described in Section 4.6.6.

Element-width selection. All operation classes select their element width at runtime via the two-bit `sew_i` input (`2'b00`=8-bit, `2'b01`=16-bit, `2'b10`=32-bit). Internal generate loops instantiate one processing slice per bus byte (16 slices for VLEN=128), and the slices are grouped into elements of the appropriate width by combinational carry chaining and MSB indexing rather than by structural replication of element- wide datapaths. This byte-slice approach allows the same hardware to process 16 eight-bit elements, 8 sixteen-bit elements, or 4 thirty-two-bit elements without any mode switching beyond the `sew_i` input.

4.8.2 Block Diagram

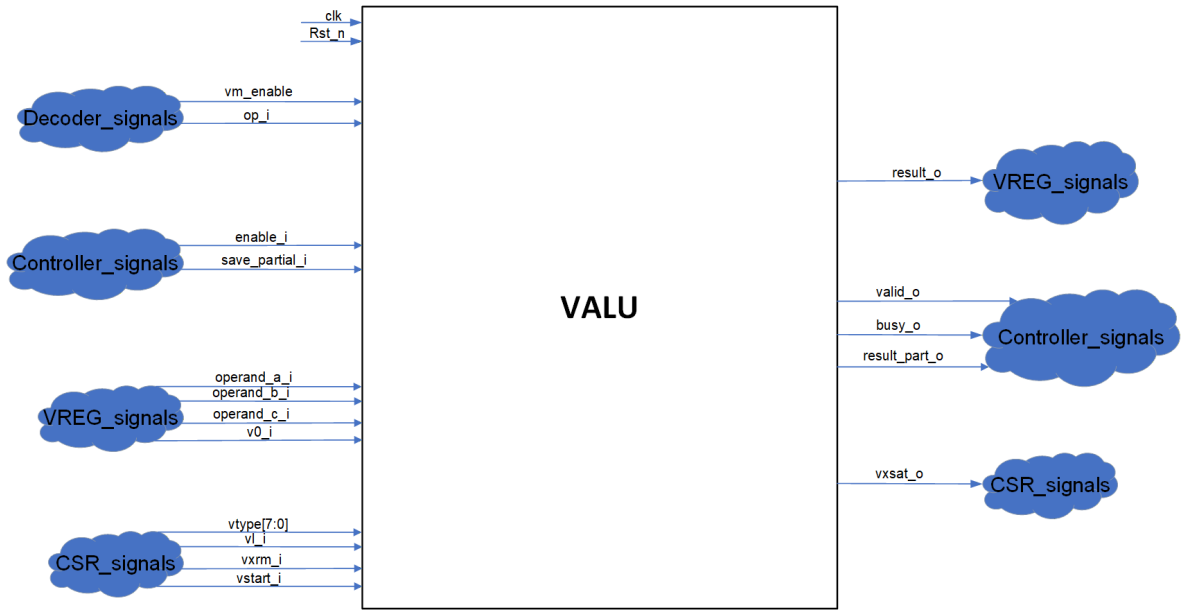

Figure 4.6: Internal combinational block diagram of the `vxt_alu` module, detailing operand selection and result computation logic.

4.8.3 Port-Level Interfaces

The ALU sits at the centre of the VXT block diagram (Figure 4.1) and exposes five distinct interface groups, summarised in Table 4.16.

Table 4.16: Port-level interfaces of `vxt_alu`, grouped by source/destination block.

Group	Signal	Dir	Description
Decoder interface			
	`op_i` (`alu_op_e`)	In	Enumerated ALU operation code (e.g. `ALU_VADD`, `ALU_VWMUL`).
	`unit_i`, `mode_i`	In	Execution unit selector and arithmetic mode (`MODE_ARITH`, `MODE_WIDEN`, `MODE_MOVE`).
	`operand_i` (`operand_e`)	In	Operand form: IVV, IVX, IVI, MVV, or MVX.
	`widen_i`, `narrow_i`	In	Asserted for widening and narrowing instructions respectively.
	`mask_comp_i`, `mask_logic_i`	In	Distinguish mask-comparison (`vmslt`, `vmseq`) from mask-logic (`vmandn`, `vmnand`) operations.
	`vm_i`	In	Mask-enable bit (active-low of `instr[25]`); when clear, all elements are written.
	`funct3_i`[2:0], `imm5_i`[4:0]	In	Raw `funct3` field and 5-bit immediate, consumed by the operand-B mux.

Continuation of Table 4.16			
Group	**Signal**	**Dir**	**Description**
	use_imm_i	In	Asserted for OPIVI-form instructions to select imm5_i as the second ALU operand.
Register file interface			
	operand_a_i [VLEN-1:0]	In	vs2 contents; primary vector operand for the selected operation.
	operand_b_i [VLEN-1:0]	In	vs1 contents, scalar/immediate broadcast, or second vector operand selected by the operand-B mux.
	operand_c_i [VLEN-1:0]	In	Current vd contents, fed back for accumulate, tail, and mask policy inside result_o.
	result_o [VLEN-1:0]	Out	Computed element-wise result, with tail and mask policy already applied.
Scalar core interface			
	x_operand_i [XLEN-1:0]	In	Scalar rs1/rs2 value forwarded for IVX/MVX-form operations.
	x_result_o [XLEN-1:0]	Out	Sign-extended scalar result for vmv.x.s and reduction instructions.
Controller interface			
	is_mul_i	In	Asserted for vmul, vmulh, vwmul, vwmacc; enables multi-cycle sequencing.
	is_reduction_i	In	Asserted for vredmax.vs and vredsum.vs.
	is_vmv_i, is_vmovnr_i, is_vmv_xs_i, is_vmv_imm_i	In	Move-class sub-type flags for vmv.v.*, whole-register move, vmv.x.s, and immediate-broadcast forms.
	alu_valid_o	Out	Asserted when a valid result is available on result_o/x_result_o.
	alu_busy_o	Out	Asserted while a multi-cycle operation (multiply, reduction) is in progress.
CSR unit interface			
	vsew_i	In	Selected element width, used to configure the operand-slicing and result-packing datapath.
	vl_i, vl_computed_i	In	Active vector length; determines which elements are computed versus held at tail value.
	vstart_i	In	Start index; elements below vstart_i are passed through unmodified from operand_c_i.

Continuation of Table 4.16			
Group	**Signal**	**Dir**	**Description**
	`vta_i`, `vma_i`	In	Tail-agnostic and mask-agnostic policy bits applied when composing `result_o`.
Reduction interface			
	`redop_o`[1:0]	Out	Reduction accumulator select for `vredmax.vs`/`vredsum.vs` sequencing.
	`redresult_i [VLEN-1:0]`	In	Partial reduction accumulator fed back into the ALU for the next cycle.

The `result_mux` interface deserves particular note: rather than having every downstream consumer of the computed result re-derive which lane or accumulator to use, the ALU resolves this once and exports a single select signal, keeping the result-selection logic in exactly one place.

4.8.4 Operand-B Multiplexer (`op_b_mux`)

Before any arithmetic can begin, the second operand must be resolved to a VLEN-wide bus in the same format as operand A. The `op_b_mux` module is a purely combinational block that performs this resolution in a single cycle, freeing every downstream arithmetic slice from any awareness of instruction encoding.

Three operand-source paths. The multiplexer selects among three sources based on the `sel_b_i` encoding produced by the decoder (Section 4.3.4):

- **Vector-register path (`.vv`).** The full VLEN-wide `operand_b_v_i` bus is passed through unchanged. This path requires no width conversion because the register file already delivers data in the correct format.
- **Scalar-register path (`.vx`).** The XLEN-wide scalar value `operand_b_x_i` is truncated to the active element width (low 8, 16, or 32 bits depending on `sew_i`) and replicated across all element lanes to fill the 128-bit bus. This broadcast operation (sometimes called a splat) ensures that every element of the vector sees the same scalar operand in the position it expects.
- **Immediate path (`.vi`).** The 5-bit signed immediate `operand_b_imm_i` is sign-extended to the active element width and then replicated in the same manner as the scalar path. The sign extension respects the RVV specification's rule that vector immediates are treated as signed integers regardless of the operation.

The output of `op_b_mux` is connected directly to the `operand_b_i` port of `vxt_alu`, so from the arithmetic datapath's perspective there is always a uniform VLEN-wide second operand bus regardless of the instruction's operand form. The `op_b_mux` instance is the only place in the design where the distinction between `.vv`, `.vx`, and `.vi` instruction forms has any hardware significance.

4.8.5 Main Arithmetic Datapath (`vxt_alu` Top-Level)

The `vxt_alu` top-level module implements all single-cycle operation classes. It is organised as a set of parallel computation paths, each producing a candidate result bus, with a final priority multiplexer selecting among them based on the decoded operation code.

Byte-slice arithmetic and carry chaining. The most widely used path, integer arithmetic, instantiates one processing slice per byte via a generate loop. Each slice performs an 8-bit add with carry-in from the previous slice in the same element; slices at element boundaries receive an initial carry that depends on the operation (subtraction injects a carry-in of 1; addition injects 0). This ripple-carry arrangement across the bytes of an element means that a 32-bit addition, for example, is formed naturally by four 8-bit slices chained together, without any separate 32-bit adder. Comparison, min, max, and saturating add all reuse the same adder slices, deriving their results from the carry-out and sign bits of the element-width MSB slice.

Operation classes. Table 4.17 summarises the eight operation classes handled in the top-level datapath.

Class	Representative instructions	Implementation note
Integer arithmetic	`vadd`, `vsub`, `vrsub`, `vmin`, `vmax`	Byte-slice adder with carry chain; min/max derived from comparison carry-out.
Saturating arithmetic	`vsadd`	Same adder; overflow detected at MSB byte of each element; result clamped to `0x7F...` or `0x80...`; sets `vxsat_o`.
Shift	`vsrl`, `vsra`, `vsll`	Per-element barrel shift; shift amount masked to $\log_2$(SEW) bits per the RVV specification [2].
Widening/Narrowing	`vnsrl`, `vnsra`, `vnclip`	Narrowing shifts halve the element width and pack results; `vnclip` additionally applies rounding and saturation.
Mask-output compare	`vmslt`, `vmseq`, `vmsle`	Result is a 1-bit-per-element mask packed into the low bits of `result_o`; inactive bits set to 1.
Mask logic	`vmnand`, `vmandn`	Bitwise operations on mask registers; always single-beat.
Merge/Move	`vmerge`, `vmv.v.x`, `vmv.v.i`	Select between operand A and operand B per element using `v0` or unconditionally.
Index generation	`vid.v`	Writes element index i into element slot i; implemented as a generate of compile-time constants.

Table 4.17: Operation classes implemented in the `vxt_alu` top-level datapath. Multiply-class and reduction operations are handled by the `vxt_mul` and `vxt_reduction_tree` sub-modules respectively (Sections 4.8.6 and 4.8.7).

Policy application at the output stage. VTA, VMA, and masking are applied once, at the final output multiplexer, rather than being re-implemented in each computation path. For each byte position i in the 128-bit result bus, the output mux selects among four values using the same priority ordering as the VLSU merge logic (Section 4.7.7):

1. Prestart (`element_index` $<$ `eff_vstart`): retain `old_vd_i`.
2. Tail (`element_index` $\geq$ `eff_vl`): retain `old_vd_i` if `vta=0`, or write `0xFF` if `vta=1`.
3. Active, masked-off (`vm_enable_i=1`, `v0[element]`$=0$): retain `old_vd_i` if `vma=0`, or write `0xFF` if `vma=1`.
4. Active, unmasked: use the arithmetic result.

The `eff_vl` and `eff_vstart` values are re-derived from the raw `vl_i`, `vstart_i`, and `lmul` inputs using the same beat-offset arithmetic described in Section 4.7.9, ensuring that the correct

element indices are applied on each beat of a LMUL=2 instruction without any additional state in the ALU.

4.8.6 Multiplier Sub-Module (vxt_mul)

Multiply operations are separated into vxt_mul because they require multiple clock cycles and contain their own registered pipeline state. The module handles four instruction variants: vmul (non-widening, lower-half product), vmulh (non-widening, upper-half product), vwmul/vwmulu (widening, signed or unsigned), and vwmacc (widening multiply-accumulate).

Start/busy/valid handshake. The controller fires enable_i once on the cycle it enters ST_ALU_EXEC. vxt_mul latches all operands on that edge, asserts busy_o, and then deasserts busy_o and asserts valid_o when the result is ready. The exact number of cycles between enable_i and valid_o depends on the operation and element width:

- **SEW=8 and SEW=16, all non-accumulate variants**: the multiplier completes in the same cycle as enable_i (fast path), so busy_o is never asserted and the controller's latency is identical to a single-cycle ALU operation.
- **SEW=32, non-widening**: one additional cycle of pipeline latency; busy_o is asserted for that cycle and deasserted when valid_o fires.
- vmulh **SEW=32**: two additional cycles, since the upper-half product requires an extra accumulation stage.
- vwmul/vwmulu: the widening result is produced as two sequential VLEN-wide beats. result_part_o=1 accompanies the upper-half beat (written to vd+1) and result_part_o=0 accompanies the lower-half beat (written to vd), in that order.
- vwmacc: the same two-beat widening protocol, with an additional cycle consumed to read the existing destination register value before accumulation begins (Section 4.6.6).

Policy enforcement inside vxt_mul**.** Unlike the single-cycle datapath, where policy is applied at the top-level output mux, vxt_mul applies its own element-level policy internally. This is necessary because the multiplier pipeline registers its operands and produces results over multiple cycles; applying policy at the top-level output mux would require also registering old_vd_i, which would introduce an additional pipeline stage on a path that the controller does not account for. Instead, vxt_mul receives vd_i (the current destination register contents), v0_i, vta_i, vma_i, vl_i, and vstart_i as direct inputs and applies the same four-priority element-output selection as the top-level datapath, independently for each result beat.

4.8.7 Reduction Tree (vxt_reduction_tree)

The reduction tree implements vredsum.vs (sum of all active elements) and vredmax.vs (signed maximum of all active elements), placing the scalar result into element 0 of the destination register vd. Because VXT supports LMUL=2, the reduction must be capable of spanning two 128-bit register beats while accumulating a single running result.

Combinational tree and registered output. The core of the module is a fully combinational binary tree that reduces the active elements of the current beat's `vs2` operand into a single 32-bit value. Each element slot is first normalised: sign-extended to 32 bits if active (within `[vstart, vl)` and not masked off), or replaced by a neutral element (0 for sum, -2^{31} for signed maximum) if inactive. The tree then reduces these normalised values with either addition or signed-maximum comparisons, seeded from the scalar value in element 0 of `vs1` (the accumulation seed per the RVV specification [2]). A registered output latch (`result_r`) captures the combinational tree's output on the positive clock edge, eliminating any long-path timing hazard across the register file write-back boundary.

Two-beat LMUL=2 accumulation. For LMUL=2, the controller asserts `save_partial_i` on beat 0 of the reduction. This causes the reduction tree to capture its beat 0 result into a registered `partial_res` register. On beat 1, the combinational tree is seeded with `partial_res` rather than the `vs1` scalar, effectively chaining the two beats. The final result in `result_r` after beat 1 is the reduction across all active elements of both register halves. The controller's `alu_save_partial_o` signal (Section 4.6.6) is the exclusive mechanism by which `save_partial_i` is driven; no other path asserts it.
The reduction result is assembled into a VLEN-wide bus (`reduction_vec_res`) by placing the 32-bit scalar in element-0 position and filling remaining element slots with the pre-existing contents of `old_vd_i`, consistent with the RVV rule that only element 0 of the destination is written by a reduction instruction.

4.8.8 Design Notes and Known Limitations

`vmv.v.v` **not implemented.** The whole-register copy form that moves one vector register directly into another using the ALU datapath (`vmv.v.v`) is not implemented. The decoder marks this encoding as invalid and routes whole-register copies through the dedicated `vmovnr` / register-file-bypass path in the controller instead (Section 4.6.3). This choice avoids adding a passthrough case to the ALU output mux for an operation that the controller already handles more efficiently without ALU involvement.

SEW=64 not supported. The byte-slice arithmetic and carry-chain structure supports 8-bit, 16-bit, and 32-bit element widths. Extending to SEW=64 would require widening the carry chain across 8 bytes per element and re-deriving all element-index constants in the generate loops. This extension falls outside the VLEN=128, XLEN=32 design envelope established in Chapter 3 and is not implemented.

4.9 RVV Policy Implementation Across the Datapath

The RISC-V Vector specification [2] defines three element-level execution policies that every compliant implementation must enforce: the prestart policy governed by `vstart`, the tail policy governed by `vta`, and the mask policy governed by `vma`. These are not features of a single block; they are cross-cutting behavioural commitments that each block in VXT contributes to jointly. This section traces how each policy is upheld across the CSR unit, the decoder, the controller, and the VLSU, and concludes with a description of how the mask register is indexed consistently across multi-beat operations.

4.9.1 `vstart`: Prestart Elements

The `vstart` CSR holds the index of the first element that a vector instruction is permitted to modify. Every element whose index falls below `vstart` is a *prestart* element: it must be left entirely untouched by the instruction, as though the instruction had never run for that element. This prestart guarantee is the mechanism by which the specification allows an interrupted vector instruction to be restarted cleanly from a known mid-vector position after a precise trap.

The CSR unit's handling of `vstart` is described in Section 4.4.7: the register is cleared to zero on every cycle by default, holding a non-zero value only when a standard CSR instruction has explicitly written it. For the vast majority of instructions (every instruction that completes without an interruption) `vstart` is zero on the cycle the instruction executes, meaning there are no prestart elements and the policy imposes no constraint.

When `vstart` is non-zero, the VLSU is the block responsible for honouring the prestart exclusion. For unit-stride loads, the byte-enable mask presented to the OBI bus is computed per-element and the lanes corresponding to prestart elements are cleared, so those bytes are never fetched from memory; the corresponding positions in the output data are filled from the previous register contents rather than from memory data. For unit-stride stores, the same byte-enable logic clears the lanes for prestart elements, preventing any write to memory for those positions. For strided loads, the element-iteration counter is seeded at `vstart` rather than at zero, so no memory transaction is issued for any prestart element at all and the output accumulator is pre-loaded with the existing register contents so those slots are already correct without any further action.

The ALU does not see a separate `vstart` enforcement path because all ALU policies (prestart, tail, and mask) are enforced through the same per-element selection mux that the tail and mask policies also use, driven by the live `vstart` value from the CSR unit. The division between ALU-side enforcement for arithmetic operations and VLSU-side enforcement for memory operations is clean and without duplication.

4.9.2 `vta`: Tail Agnostic

Elements beyond the active vector length, indices from `vl` up to the maximum elements per register group, are *tail* elements. The `vta` bit in `vtype` determines how these elements are treated. Under the undisturbed setting (`vta=0`), tail elements must retain exactly the value they held before the instruction executed. Under the agnostic setting (`vta=1`), the implementation may write any value to tail elements; VXT writes the all-ones byte pattern (`0xFF` per byte) as its canonical agnostic fill.

The `vta` bit is a live combinational output of the CSR unit, available every cycle without any latency. The controller passes it to the ALU and to the VLSU on every cycle without registering it, ensuring that both execution units always operate against the current tail policy. The ALU applies the policy per-element within its writeback selection mux: for each element position it determines whether the position is active or tail based on the current `vl`, and selects the appropriate value, computed result or preserved old value or `0xFF` fill, accordingly. The VLSU applies the same distinction within its load-merge logic: for unit-stride loads, the per-element priority mux that selects the output byte for each position includes a tail branch that either preserves the old register byte or forces `0xFF`. For strided loads, the tail policy is applied after all elements have been accumulated in the load result, replacing tail slots with the appropriate value before the result is written to the register file. Store operations are unaffected by `vta` because stores read the register file rather than writing it.

4.9.3 `vma`: Mask Agnostic

Among the active body elements, those within the `vl` bound and beyond `vstart`, some may be *masked off* when the instruction is executed under a mask. An element is masked off when masking is enabled for the instruction and the corresponding bit in the mask register is zero. The `vma` bit determines whether masked-off elements are left undisturbed (preserved at their prior value) or may be overwritten (VXT fills them with `0xFF`).
The `vma` policy is implemented by the same per-element selection mechanism as `vta`, with a lower priority: a masked-off element is only reached by the mask branch of the mux if the element has already passed the prestart and active-range checks. The same undisturbed-or-fill choice applies. For stores, masked elements are excluded by clearing their byte lanes in the byte-enable mask, so those bytes are never written to memory regardless of the `vma` setting.
Because `vta` and `vma` are both sourced from the same live CSR output and both consumed by the same per-element selection logic in the ALU and VLSU, there is no possibility of the two policies becoming inconsistent or of either block operating against a stale copy of either bit.

4.9.4 Mask Register Indexing Across LMUL Beats

When masking is enabled, the architectural mask register `v0` supplies one bit per element slot across the full maximum element count. For single-register-group operations this is straightforward: the first bit covers element 0, the second covers element 1, and so on up to the maximum elements per beat. For two-register-group operations the mask register must be indexed differently across the two beats: beat 0 uses the lower half of the mask bit-vector and beat 1 uses the upper half.
This beat-offset indexing is what the VLSU documentation terms the *global element base*: a beat-dependent offset that locates the correct region within the mask register for the current beat. For beat 0 the base is zero, so element i of beat 0 consults mask bit i. For beat 1 the base equals the number of elements per beat, so element i of beat 1 consults mask bit $\text{elements_per_beat} + i$. This re-indexing is computed entirely within the VLSU's effective-parameter block and is transparent to the register file, the controller, and the ALU.
For the ALU, the mask register is supplied by the register file's dedicated mask-output port, which continuously presents the full 128-bit contents of register `v0` (Section 4.5.2). The beat index is forwarded to the ALU by the controller (Section 4.6.7), and the ALU extracts the appropriate bit per element using the same global-element-base calculation. The net effect is that all four blocks, the CSR unit (which holds `vma` and `vstart`), the controller (which supplies beat indices and policy bits), the VLSU (which enforces policies for memory operations), and the ALU (which enforces policies for arithmetic operations), each contribute exactly the part of policy enforcement that falls within their natural scope. No block duplicates another's responsibility, and no additional inter-block handshaking is needed to coordinate policy application.

4.10 Arbitration and Bus Integration

4.10.1 Why Arbitration Is Needed

VXT is attached to the CV32E40P scalar core as an application processing unit rather than as a separate bus master with its own dedicated memory port. As a consequence, both the scalar core's load/store unit and VXT's VLSU need to access data memory over the same OBI data bus. In the absence of arbitration, a vector memory instruction arriving while the scalar core is completing a data transaction would cause both masters to drive the bus simultaneously, corrupting the transaction in progress.

The arbitration problem is particularly acute because the scalar core's stall-while-vector-executes model (Section 4.6.3) does not fully eliminate the overlap risk: the scalar core may have already issued a data request on the bus in the same cycle that the controller begins to process a vector load or store. The controller therefore checks whether the scalar core's data bus is currently occupied before advancing any vector memory instruction, and defers for one cycle if a conflict is detected. Once a vector memory instruction begins its memory phase, the arbiter grants exclusive bus ownership to the VLSU for the duration of that instruction.

4.10.2 Arbitration Scheme

The arbiter is implemented as a simple two-input, one-output combinational multiplexer on the OBI data channel, controlled by a single ownership select signal driven by the controller. When the controller is not executing a vector memory instruction, the select signal routes the scalar core's data-channel signals through to the memory subsystem unmodified. When the controller enters its memory-execution phase, and for the initial transaction issue that occurs one cycle earlier as part of the early-start optimisation, the select signal switches ownership to the VLSU, and the scalar core's data-channel outputs are masked off. The OBI grant signal returned to the scalar core is also suppressed during this window, which back-pressures the scalar pipeline naturally without requiring any additional stall logic in the core itself.
The controller asserts the bus-ownership signal at the beginning of a vector memory operation and holds it for every cycle of the memory phase, including across all beats of a two-register-group memory instruction. It is deasserted in the RETIRE cycle, restoring scalar bus access exactly when the vector operation is known to be complete. Because RETIRE is always one cycle long (Section 4.6.3), the handover back to the scalar core is deterministic and requires no additional arbitration protocol.

4.10.3 Timing Stability Guarantees

The OBI protocol used by the CV32E40P ecosystem imposes a stability requirement on the requesting master: address and control signals must remain stable from the cycle they are first presented until the cycle the grant is received. This property, sometimes denoted TRANS_STABLE, ensures that a memory subsystem that cannot grant immediately, because a higher-priority request is in flight, or because the memory itself has a multi-cycle latency, can safely hold off the grant without losing the address information.
The VLSU satisfies this requirement structurally rather than through additional buffering. For unit-stride operations the base address is a constant input held stable for the lifetime of the instruction, so it is trivially stable across any number of grant-wait cycles. For strided operations the address for the current request window is derived from a counter that advances only when a grant is received, meaning the address value is guaranteed not to change while the grant is outstanding. As a result, the VLSU never needs to latch or re-drive a transaction address to meet the stability requirement; the address computation logic holds its value unchanged between request and grant.
The scalar core's transactions are also compliant with the same stability requirement by construction, since the CV32E40P's OBI data-channel implementation was designed to the same protocol. The arbiter therefore does not need to introduce any buffering or retiming on either of its two input channels; it is a pure combinational select between two protocol-compliant sources, and whichever source is currently selected always presents a stable, valid OBI transaction.

4.11 Key Design Decisions and Trade-offs

This section collects the five most consequential design decisions made during the implementation of VXT, each of which reflects a trade-off between competing objectives, typically between simplicity and correctness on one side, and performance and generality on the other.

Negative-edge clocking for the register file. The register file's write port clocks on the falling rather than the rising clock edge. This creates a natural half-cycle margin between the point at which the controller produces a write-enable decision (on the rising edge) and the point at which that decision takes effect in the register file (on the falling edge). Without this margin, the write-enable signal would need to be presented with zero propagation budget, which is not achievable in a purely combinational decode path. The trade-off is that the register file does not behave as a standard synchronous RAM; the write data must be stable by the negative edge rather than the positive edge of the clock.

Structural generate blocks rather than loop constructs in combinational logic. All combinational logic that operates over arrays of elements, per-element active flags, byte-enable masks, load-merge results, and strided-load accumulators, is implemented using generate blocks that unroll the element loop at elaboration time. This means every element slot is represented by an explicitly named logic path in the elaborated netlist, rather than being inferred from a runtime loop index. The benefit is that synthesis tools can optimise each slot individually, there are no implied sensitivity-list issues that can cause simulation mismatches, and the elaboration is deterministic across different simulators and synthesis environments. The cost is verbosity: the RTL is larger than it would be with loop constructs, and any modification to the element-count parameter requires the generate bounds to be re-verified.

Removal of strided stores. The original specification included both strided loads and strided stores. Strided stores were subsequently removed from the implemented instruction subset. A strided store requires writing individual elements to non-contiguous memory locations, which demands a separate per-element address computation for each element and a data-rotation mux to align each element to byte lane zero of the OBI bus before the transaction is issued. Both of these additions sit on timing-critical paths and add significant combinational depth to the VLSU's address-generation and store-data paths. Because strided stores appear rarely in the AI inference workloads that VXT targets: matrix and convolution kernels are predominantly read-dominated once the weights are loaded, the simplification was judged to be correct for the design's application envelope. The strided load path, which does not require per-element address generation (it batches elements from a single wide fetch), is retained.

Width conventions: one-bit-wider element count representations. The active vector length `vl` and the per-beat effective vector length and start values are all declared one bit wider than might seem necessary. The reason is that `vl` is a count, not an index: its maximum value equals the maximum number of elements, which for 8-bit elements at VLEN=128 is sixteen. The value sixteen does not fit in a four-bit field (which can hold indices 0 through 15) but does fit in a five-bit field (which can hold values 0 through 31). Carrying the extra bit throughout the comparison and arithmetic on element counts eliminates a class of off-by-one and overflow bugs that would otherwise require careful per-use qualification. Similarly, explicit zero-extension is used whenever a narrower element-count value is promoted to a wider context, avoiding any reliance on implicit sign-extension behaviour that could differ between tools.

Non-pipelined execution model. VXT executes one vector instruction at a time, in order, with no chaining, no instruction overlap, and no speculative execution. The scalar core stalls for the entire duration of each vector instruction, and the next instruction is not accepted until `instr_done` is asserted. This model sacrifices throughput, a pipelined or chained design could in principle begin the next instruction before the current one has retired, in exchange for a verification surface that is dramatically smaller than that of a concurrent design. In the context of a first-generation AI accelerator where correctness of the vector result is more important than peak throughput, and where the main performance leverage comes from operating on sixteen elements in parallel rather than from instruction-level overlap, the non-pipelined model is the appropriate choice. The trade-off is explicitly acknowledged: future work that increases VLEN, adds wider LMUL support, or targets higher clock frequencies would need to revisit this decision.

4.12 Summary

This chapter has provided a bottom-up description of every block in the VXT microarchitecture, proceeding from the instruction boundary at the top-level wrapper through the decoder, CSR unit, register file, controller, ALU, VLSU, and policy-enforcement layer, and concluding with the OBI arbitration scheme and the key design decisions that shaped the implementation.

The central organizing principle that runs through all of these blocks is the division of responsibility between combinational and sequential elements: the decoder, CSR unit, and register-file read paths are purely combinational and present their outputs within a single propagation delay of their inputs; the controller, register file write port, and VLSU sequencer are the only stateful blocks; and the ALU sits between the two, with purely combinational datapath logic wrapped in a registered multiply pipeline for the multi-cycle cases. This organisation means that the latency of any instruction is determined almost entirely by the FSM path through the controller, which in turn is determined by the instruction class and LMUL grouping as summarised in Table 4.13.

The ISA subset implemented by VXT, configuration instructions, integer arithmetic and shift, multiply and widening multiply, reduction, mask operations, register-group move, and unit-stride and strided load, was selected in Chapter 3 to cover the core compute patterns of AI inference at 8-, 16-, and 32-bit element widths. The microarchitecture described here realises that subset faithfully: every instruction class in the supported subset has a corresponding path through the controller FSM, a corresponding operation code in the ALU, and a corresponding set of CSR fields. The three RVV element policies, prestart under `vstart`, tail under `vta`, and mask-off under `vma`, are enforced jointly across the CSR unit, the ALU, and the VLSU, as described in Section 4.9, with no single block bearing sole responsibility for any policy and no policy left unenforced.

The design passes the full functional verification suite described in Chapter 6, including exhaustive coverage of the `vmovnr` write-enable timing fix, the three-case `vl` computation algorithm, and the VTA/VMA policy for all supported element widths and LMUL configurations. Chapter 6 describes the testbench architecture and the coverage metrics achieved.

Chapter 5

Verification and Validation Methodology

Contents

5.1 Introduction 100
5.2 Verification Objectives 101
5.3 Overall Verification Strategy 101
5.4 Instruction-Level RTL Verification 102
5.5 Kernel and Model Verification Methodology 103
5.6 Software Build Flow for RTL Simulation 104
5.7 Golden Reference Generation Using QEMU 104
5.8 RTL Simulation and Memory Dump Collection 105
5.9 Automated Output Comparison 106
5.10 Debug and Failure Diagnosis 107
5.11 Validation Results Summary 108
5.12 Limitations of the Verification Methodology 110
5.13 Chapter Summary 110

5.1 Introduction

The starting point of this project was the CV32E40P scalar core, with the goal of building a custom vector extension (referred to throughout this thesis as VXT) to accelerate AI inference workloads. A custom hardware extension of this kind carries a verification burden that an off-the-shelf, pre-validated core does not: every block, every instruction, and every integration point between the new vector unit and the existing scalar pipeline is a potential source of functional error that no vendor test suite has ever exercised. Without a deliberate verification methodology, a hardware bug and a software bug can be indistinguishable from a single failing test, and a project of this scope, spanning ISA design, RTL microarchitecture, and an AI kernel software library, has no shortage of places for such bugs to hide.

This chapter presents the verification and validation methodology developed and applied throughout the project. Two related but distinct activities are involved. *Verification* asks whether each piece of hardware, an isolated functional block, a single vector instruction, the fully integrated VXT-augmented core, behaves according to its specification. *Validation* asks the higher-level question of whether complete AI kernels and model stages, running as software on that hardware, produce the correct numerical result. A kernel cannot validate correctly if

the instructions it issues are not first verified, and instruction-level verification alone cannot guarantee that a multi-instruction kernel composes those instructions correctly. The chapter therefore follows the same bottom-up structure that was used during development, beginning with isolated RTL verification and progressing upward to complete AI kernels and, ultimately, full model-stage validation. The chapter documents both the tooling developed along the way and the evidence that the vector extension and the software kernels running on it are functionally correct.

5.2 Verification Objectives

The verification process described in this chapter is designed to establish the following:

- Each supported vector instruction produces the result expected by the RISC-V Vector Extension specification, when executed on the implemented hardware.
- The VXT vector unit operates correctly once integrated with the CV32E40P scalar core, including correct instruction sequencing and data flow between the scalar and vector domains.
- Vectorized AI kernels produce functionally identical output to their scalar counterparts, for the same input data and the same algorithm.
- Hardware simulation output can be compared, byte-for-byte, against an independent golden software reference, rather than relying on visual inspection or spot-checking.
- Failures can be attributed to a specific stage of the pipeline, RTL behavior, software-kernel implementation, memory mapping/addressing, or test-setup error , rather than surfacing only as an undifferentiated "output is wrong."

5.3 Overall Verification Strategy

The verification strategy adopted in this project is bottom-up: confidence is built one abstraction level at a time, and no level is trusted until the level beneath it has already been confirmed correct. Concretely, the project followed four stages:

1. **Instruction-level RTL verification.** Individual VXT blocks, and subsequently individual vector instructions, were exercised in isolation.
2. **Kernel-level verification.** Complete AI kernels, activation functions, pooling, convolution, matrix multiplication, were implemented in both scalar and vector form and compared.
3. **Model-stage verification.** The same scalar-versus-vector methodology was reused for higher-level model stages, extending confidence from individual kernels to multi-kernel pipelines.
4. **Three-way output comparison.** At both the kernel and model-stage levels, the scalar hardware output, the vector hardware output, and an independent QEMU golden reference are compared against one another, rather than against a single assumed-correct source.

The same methodology (scalar-vs-vector-vs-golden comparison) was deliberately reused unchanged between standalone kernels and higher-level model stages, so that confidence gained validating one kernel transfers directly to validating a model built from many such kernels. Figure 5.1 illustrates this layered structure: each tier builds on the confidence established by the tier below it, culminating in overall confidence in the complete system.

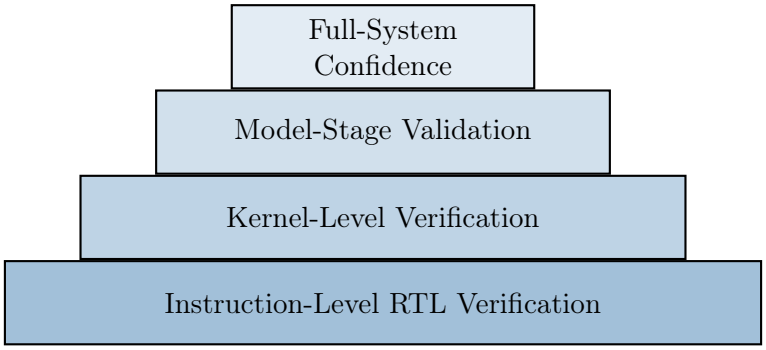

Figure 5.1: Bottom-up verification pyramid: confidence accumulates from instruction-level RTL verification up to full-system confidence.

5.4 Instruction-Level RTL Verification

The lowest verification tier targets the VXT hardware blocks and individual vector instructions in isolation, before any software kernel is involved. Two sub-stages were used.

Isolated block testing. Each VXT functional block, including the vector ALU, the register file, the load/store unit, and the controller, was first tested standalone in QuestaSim. At this stage, instructions were driven directly into a block under test using manually specified opcodes, deliberately bypassing the instruction decoder. This isolates a genuine functional bug in, for example, the ALU's arithmetic datapath from a bug in the decoder that merely fails to produce the correct control signals for that ALU, a distinction that would be impossible to draw if the decoder were already in the loop.

Integrated sequence testing. Once every block had been confirmed correct in isolation, the VXT blocks were integrated with one another and then with the CV32E40P core as a whole. The fully integrated design was exercised using ordered sequences of instructions, confirming that control and data correctly flow between the scalar pipeline and the vector unit, including the decoder, which is exercised for the first time at this stage, rather than only confirming that each block behaves correctly in artificial isolation.

Table 5.1 summarizes the instruction families supported by VXT and exercised by this verification tier. Individual mnemonic variants (immediate, scalar, and vector-vector forms) are grouped by family rather than listed exhaustively, consistent with the instruction subset documented in Chapter 6.

Figure 5.2 summarizes the instruction-level verification flow.

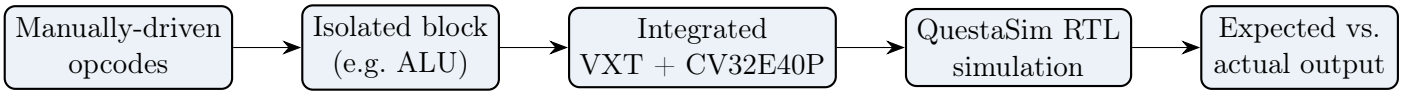

Figure 5.2: Instruction-level RTL verification flow: from manually-driven opcodes on an isolated block, through full integration, to a QuestaSim-checked result.

Table 5.1: VXT instruction families exercised during instruction-level RTL verification.

Family	**Representative Instruction**	**SEW / LMUL**	**Status**
Configuration	`vsetvli`	–	Verified
CSR access	`csrrw`, `csrrs`	–	Verified
Load / Store	`vle.v`, `vse.v`	8/16/32, 1–2	Verified
Strided Load / Store	`vlse.v`, `vsse.v`	8/16/32, 1–2	Verified
Move / Broadcast	`vmv.v.x`, `vmv.s.x`	8/16/32, 1–2	Verified
Reduction	`vredsum.vs`, `vredmax.vs`	8/16/32, 1–2	Verified
Integer Arithmetic	`vadd.vv`, `vsadd.vx`	8/16/32, 1–2	Verified
Min / Max	`vmax.vx`, `vmin.vv`	8/16/32, 1–2	Verified
Compare / Mask / Merge	`vmseq.vi`, `vmerge.vvm`	8/16/32, 1–2	Verified
Shift	`vsrl.vx`, `vsra.vv`	8/16/32, 1–2	Verified
Narrowing Shift / Clip	`vnsra.wx`, `vnclip.wx`	8/16 only, 1–2	Verified
Multiply	`vmul.vv`, `vmulh.vv`	8/16/32, 1–2	Verified
Widening Multiply / MAC	`vwmul.vv`, `vwmacc.vv`	8/16 only, 1–2	Verified

5.5 Kernel and Model Verification Methodology

With the VXT hardware itself verified at the instruction level, the next tier validates the AI kernel software that runs on it. Every kernel benchmarked in Chapter 6 was implemented in two parallel versions:

- A **scalar C version**, using only the CV32E40P's native scalar instruction set.
- A **vector C version**, using RVV assembly intrinsics targeting the VXT extension.

Both versions implement the same algorithm and are given identical input data. The scalar version serves two purposes: it is the hardware-versus-software performance baseline used throughout the benchmarking chapters, and (for the purposes of this chapter) it is also a second, independent functional implementation against which the vector version's correctness can be cross-checked. The vector version is the one under test: if it produces the same output as the scalar version, and both agree with an independent golden reference, the custom vector acceleration path is functioning correctly for that kernel. Higher-level model stages (Chapter 7) reuse this same dual-implementation methodology, with the scalar and vector model code feeding into the identical compilation, simulation, and comparison flow described in the remainder of this chapter.

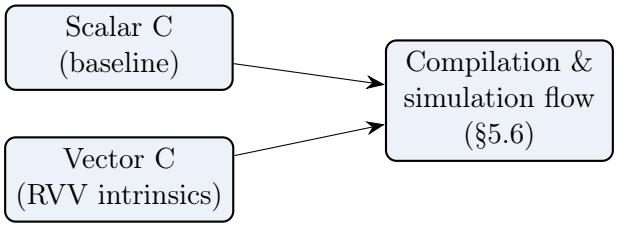

Figure 5.3: Scalar and vector implementations of the same algorithm feed into the identical compilation and simulation flow.

5.6 Software Build Flow for RTL Simulation

Both the scalar and vector C programs were cross-compiled for the RISC-V target using the xPack GNU RISC-V GCC toolchain, linked against the OpenHW runtime files, the linker script, startup file, and system-call support file (`syscalls.c`), that provide the minimal bare-metal environment expected by QuestaSim's testbench. Compilation produces two artifacts that are both consumed downstream: an Intel-hex memory image and a disassembly listing. The hex image supplies the raw instruction bytes loaded into `firmware.dat` and read into simulated instruction memory at the start of simulation; the disassembly listing supplies the instruction addresses that the testbench uses to count clock cycles (for example, to detect when execution reaches a known address such as the program's exit point).
The following table summarizes the role of each tool in this flow.

Table 5.2: Tools used in the software build flow for RTL simulation.

Tool	**Role**	**Output File**
xPack GNU RISC-V GCC	Cross-compilation and linking	ELF binary
QuestaSim	RTL simulation	Simulation transcript
Python scripts	Test-data generation, automation	Various

Figure 5.4: Software build flow: from C source to instruction and address data loaded into RTL simulation.

The location of each program's output array in memory is determined by the linker. In the project's earliest manual workflow, this address was wherever the linker happened to place it, which differs from kernel to kernel and even between the scalar and vector build of the *same* kernel. Locating it required reading the disassembly listing by hand for every single test, a process revisited in Section 5.8, where it became the motivation for a deliberate change to the linker script itself.

5.7 Golden Reference Generation Using QEMU

A scalar-versus-vector comparison alone confirms that the two hardware implementations *agree with each other*; it cannot, on its own, rule out the possibility that both implementations share the same algorithmic mistake and agree on a wrong answer. An independent third reference is required, and for this project that reference is QEMU's RISC-V Vector Extension emulation.
A QEMU-compatible variant of each kernel's C source was produced for this purpose, distinct from the version compiled for hardware. The two differ in exactly two respects. First, the QEMU build links as an ordinary user-mode Linux binary with no custom linker script, so any output array declared `extern` for the fixed-address hardware build (Section 5.8) must instead be a local `static` array here, the symbol the hardware linker script defines simply does not exist in a plain glibc link, and the build would fail with an unresolved-reference error otherwise. Second, the QEMU build appends a loop that prints every element of the output array, one value per line, in hexadecimal:

```
for (size_t i = 0; i < sizeof(dst)/sizeof(dst[0]); i++) {
    printf("%04x\n", (unsigned short) dst[i]);
}
```

Two details in this loop matter more than they first appear. Dividing by `sizeof(dst[0])` is essential: `sizeof(dst)` alone yields the array's size in *bytes*, not its element count, and using it directly as a loop bound silently reads past the end of the array for any element type wider than one byte. The explicit cast before printing is equally essential: passing a signed value narrower than `int` to a variadic function such as `printf` triggers default argument promotion, which sign-extends a negative value before the hexadecimal format specifier ever sees it , producing, for example, eight hexadecimal digits of sign-extension garbage instead of the intended four-digit two's-complement representation. Both defects were encountered during development and are recorded in Section 5.10 as concrete examples of the methodology catching errors in the test infrastructure itself, not only in the design under test.
The resulting binary was compiled with `riscv64-linux-gnu-gcc` and executed under QEMU's user-mode RISC-V emulation with full vector-extension support enabled, redirecting its output directly to a golden reference file:

```
riscv64-linux-gnu-gcc -static -march=rv64imcv -mabi=lp64d <kernel>.c -o <kernel>
qemu-riscv64 -cpu max ./<kernel> > golden_output.txt
```

QEMU's emulation is functionally accurate to the RVV specification but has no timing model for the custom VXT pipeline. It is used here as a functional reference only. Cycle counts and performance data come exclusively from QuestaSim RTL simulation.

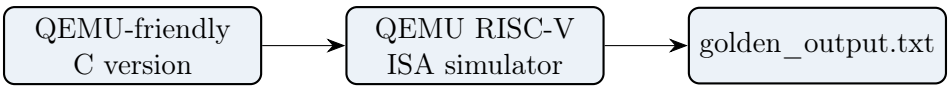

Figure 5.5: Golden reference generation: a QEMU-friendly build of the kernel is executed under QEMU's ISA-level emulation to produce the golden reference file.

In early development, the golden reference was compared against the hardware dump by hand, element by element. This was feasible for small kernels but became impractical as output sizes grew, a representative pooling configuration alone produces 4,096 output elements , and manual comparison at that scale is both slow and unreliable as a verification method. This difficulty was the direct motivation for the automated comparison pipeline described in the remainder of this chapter.

5.8 RTL Simulation and Memory Dump Collection

The scalar ELF and the vector ELF for a given kernel are simulated *separately* in QuestaSim, they are two distinct binaries, and therefore two distinct simulation runs, not two outputs of a single run. After each simulation completes, the output region is dumped from the simulated data memory using Questa's `mem save` command, producing `scalar_output.mem` from the scalar run and `vector_output.mem` from the vector run.
Collecting this dump requires knowing both where the output begins and how large it is. Initially, the output's starting address was read by hand from the disassembly listing described in Section 5.6, and the output size was read from the kernel's C source. Both values vary from kernel to kernel, and, because the linker is free to place an ordinary local array wherever it chooses, the same kernel's scalar and vector builds do not necessarily place their output arrays at the same address either.

This motivated a deliberate change to the linker script: the output array's address is now *fixed* at a single, chosen location for every kernel, rather than left to the linker's default placement. This is achieved by declaring the output array as an externally-linked symbol in the C source,

```
extern int8_t output[OUTPUT_SIZE];
```

and assigning that exact symbol name an absolute address in the linker script:

```
OUTPUT_BASE = 0x05030000;
OUT_BASE    = OUTPUT_BASE;
out         = OUT_BASE;
output      = OUT_BASE;
```

Symbol resolution between the C source and the linker script is purely a textual match: the identifier used in the `extern` declaration must be character-for-character identical to a name the linker script assigns an address to, or the link fails with an unresolved reference. Aliasing both `out` and `output` to the same address in the linker script, rather than requiring every kernel file to be renamed to one specific identifier, keeps this fix non-invasive across the existing kernel codebase. With the output address fixed, the same dump configuration applies to every kernel under this convention, and only the output size (still read from the C source) changes between tests.

Table 5.3: Memory dump files produced per kernel verification run.

Output Type	**Dump File**	**Source**
Scalar hardware result	`scalar_output.mem`	Scalar ELF, QuestaSim run
Vector hardware result	`vector_output.mem`	Vector ELF, QuestaSim run
Golden software result	`golden_output.txt`	QEMU, Section 5.7

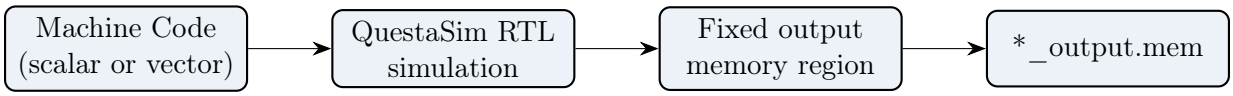

Figure 5.6: RTL simulation and memory dump collection, using the fixed output address described in this section.

5.9 Automated Output Comparison

Two pieces of automation replace the manual address lookup, manual dump command, and manual element-by-element comparison described in the preceding sections.
A `verify.do` script, invoked from the main QuestaSim `run.do` script after simulation completes, is configured with the mode of the current run (scalar or vector), the fixed output address, the output size, and the element width, and issues the corresponding `mem save` command automatically. One subtlety here is worth recording explicitly: Questa's `mem save` parses its `-startaddress`/`-endaddress` arguments as *decimal*, not hexadecimal, and silently truncates a hexadecimal-looking string at its first non-decimal digit rather than raising an error, so `verify.do` always converts the configured hexadecimal address to decimal internally before issuing the command, rather than relying on the address being typed in any particular format by hand.

A standalone Python script, `compare_outputs.py`, performs the final comparison and is run independently of QuestaSim, in an ordinary terminal, once the relevant dump files exist. It reads the scalar memory dump, the vector memory dump, and the QEMU golden reference, reconstructs the output array from each as a sequence of typed values, handling multi-byte element widths (int16, int32) by regrouping the little-endian bytes recorded in the memory dump before comparison, and performs three pairwise comparisons:

$$\text{scalar} \leftrightarrow \text{golden}, \qquad \text{vector} \leftrightarrow \text{golden}, \qquad \text{scalar} \leftrightarrow \text{vector}.$$

Verification passes only when every available comparison agrees element-for-element:

$$\text{PASS} \iff \forall\, i \in \{0, \ldots, N-1\} :\ V_{\text{scalar}}[i] = V_{\text{vector}}[i] = V_{\text{golden}}[i] \tag{5.1}$$

Any two of the three sources are sufficient to run a comparison, for example, comparing the vector result against the golden reference before the scalar run has even completed , and the script is explicit in its reporting about *why* a given pairwise comparison was not computed: a comparison skipped because one side was deliberately not supplied is reported distinctly from a comparison skipped because a supplied dump turned out to be unusable (see Section 5.10), since only the latter represents a genuine verification failure.

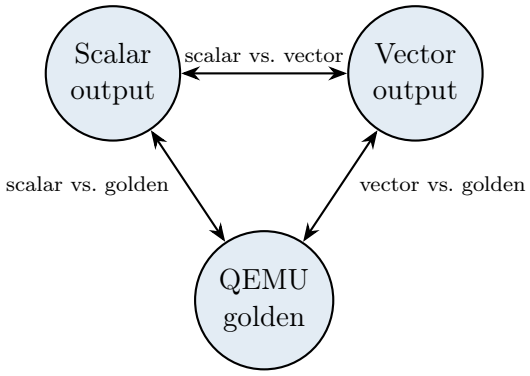

Figure 5.7: Three-way comparison: scalar hardware output, vector hardware output, and the QEMU golden reference are each compared against one another.

Table 5.4 summarizes how the outcome pattern across the three pairwise comparisons points to the likely source of a failure, the central diagnostic benefit of comparing all three sources rather than only two.

5.10 Debug and Failure Diagnosis

A methodology that cannot help diagnose failures is of limited use. Table 5.5 catalogs the failure symptoms encountered during development of this methodology, their underlying cause, and the action that resolved each one; several of these are concrete defects in the test infrastructure itself, included deliberately to illustrate that the methodology must also be capable of catching errors in its own tooling, not only in the hardware or kernel under test.
The three-way comparison of Section 5.9 is the primary mechanism for separating RTL bugs, vector-kernel implementation bugs, scalar bugs, and golden-reference or test-setup issues from one another, by examining *which* pairwise comparisons fail rather than only whether the overall result is correct.

Table 5.4: Interpreting the pattern of pairwise comparison outcomes.

Scalar vs. Golden	**Vector vs. Golden**	**Likely Meaning**
Pass	Pass	Functional validation passed for this kernel.
Pass	Fail	The vector kernel implementation or the VXT hardware path has a defect; the scalar baseline and golden reference agree, isolating the problem to the vector side.
Fail	Fail (scalar vs. vector: Pass)	Scalar and vector hardware agree with each other but both disagree with golden, investigate the golden reference or the test setup (e.g. QEMU build flags) before suspecting the hardware.

5.11 Validation Results Summary

This section summarizes the verification coverage achieved at each tier of the methodology described in this chapter. Table 5.1 (Section 5.4) already summarizes instruction-family-level coverage. Table 5.6 reports representative kernel-level verification results using the automated three-way comparison pipeline; the full kernel suite benchmarked in Chapter 6 is verified using the identical procedure, with this table populated as each kernel's verification run completes. Figure 5.8 shows the terminal output of `compare_outputs.py` from a representative kernel verification run, with all three pairwise comparisons passing across 4,096 output elements.

```
em_level/Simulation$ python3 compare_outputs.py --scalar-mem scalar_output.mem -
-vector-mem vector_output.mem --golden-hex golden_output.txt --elem-bytes 1

======================================================================
  VERIFICATION SUMMARY
======================================================================
  scalar vs golden     PASS  (4096/4096 elements match)
  vector vs golden     PASS  (4096/4096 elements match)
  scalar vs vector     PASS  (4096/4096 elements match)
======================================================================

*** OVERALL: ALL CHECKS PASSED (3/3 comparisons computed) ***
```

Figure 5.8: Terminal output of `compare_outputs.py` for a representative kernel verification run. All three pairwise comparisons, scalar vs. golden, vector vs. golden, and scalar vs. vector, pass across 4,096 output elements.

A corresponding model-stage verification summary, covering the AlexNet, Visual Wake Words MobileNetV1, and GPT-2 decoder-block stages benchmarked in Chapter 7, follows the same table structure and is completed as model-stage simulation runs conclude.

Table 5.5: Failure symptoms encountered during development and their resolution.

Symptom	**Cause**	**Debug Action**
Element count mismatch between hardware dump and golden reference, off by a factor of two	QEMU printf loop used `sizeof(dst)` (byte count) as its element-count loop bound instead of `sizeof(dst)/sizeof(dst[0])`	Corrected the golden-reference printf loop; out-of-bounds elements were also being read past the end of the array
Negative values appear with extra leading hexadecimal digits	Signed value narrower than `int` sign-extended by C's default argument promotion before reaching `printf`	Added an explicit unsigned cast of the correct width (`(unsigned short)`, etc.) before the format specifier
`mem save` dump is completely empty for the requested address range	Questa parses `-startaddress`/`-endaddress` as decimal; a hexadecimal-looking string is silently truncated at its first non-decimal digit, landing far from the intended address	Convert the configured address to decimal in `verify.do` before issuing `mem save`
Dumped range one byte longer than the actual output array	Manually computed end address as `start + size` instead of `start + size - 1` for an inclusive byte range	Standardized end-address computation in the automated tooling; confirmed against the extra trailing byte's value
`undefined reference` at link time for the output symbol	C identifier in the `extern` declaration did not exactly match the name assigned an address in the linker script	Either rename the C identifier to match, or alias the linker symbol to the name already used in the kernel source
Memory dump bytes shown as `xx` (unknown/undriven)	The design genuinely never wrote to those addresses this run, e.g. simulation `$finish` fired before the kernel finished, or a genuine RTL store-path defect	Distinguished explicitly from a comparison mismatch in the automated report, since the appropriate next step (check simulation length vs. workload size) differs from debugging a wrong numerical result

Table 5.6: Representative kernel-level verification results. S↔G: scalar vs. golden; V↔G: vector vs. golden; S↔V: scalar vs. vector.

Kernel	**Type**	**Elements**	**S↔G**	**V↔G**	**S↔V**	**Status**
Max Pooling (VWW config)	int8	4,608	Pass	Pass	Pass	Pass
Conv 11×11, i8→i32	int32	1,936	Pass	Pass	Pass	Pass
Conv 7×7, i8→i16	int16	1,024	Pass	Pass	Pass	Pass
Sigmoid (int16)	int16	8,192	Pass	Pass	Pass	Pass

5.12 Limitations of the Verification Methodology

The methodology presented in this chapter is effective but not exhaustive, and several limitations are worth stating explicitly.

- QEMU is used exclusively as a functional, ISA-level reference. It implements no timing model for the custom VXT pipeline, so it cannot be used, and is not used anywhere in this thesis, as a source of cycle-count or performance data. QuestaSim RTL simulation remains the sole source of hardware timing measurements.
- The verification approach is directed testing: specific, chosen input cases are exercised and checked, rather than exhaustively verifying every possible input. This is standard practice for a project of this scope but does not provide the exhaustive coverage guarantees of formal verification.
- Model-stage validation results are necessarily dependent on the specific test inputs and datasets selected for each model; they demonstrate correctness for the cases tested, not for every input the model could encounter in deployment.
- Comparisons must account for the numerical representation in use. Quantized fixed-point integer outputs must be compared at the correct element width and with the correct signedness, since a mismatch in either silently produces incorrect comparisons rather than an explicit error, a consideration that directly motivated several of the safeguards described in Section 5.10.

5.13 Chapter Summary

The methodology built for this project works bottom-up: block-level RTL testing first, then integrated instruction sequences, then complete kernels compared three ways against a QEMU functional reference, then full model stages. What started as a manual process, hand-reading addresses from disassembly and eyeballing output elements, was progressively automated through a fixed-address linker convention, a `verify.do` dump script, and the `compare_outputs.py` comparison tool. The same scalar-vs-vector-vs-golden comparison runs unchanged at both the kernel and model-stage level, which is what makes the results in the following chapters traceable back to a consistent verification baseline.

Chapter 6

AI/ML Primitive Design, Implementation, and Benchmark Results

Contents

6.1 Nonlinear Activation Functions . . . 113
6.1.1 Rectified Linear Unit (ReLU) . . . 113
6.1.2 Gaussian Error Linear Unit (GELU) . . . 116
6.1.3 Sigmoid . . . 119
6.1.4 Softmax . . . 121
6.2 Requantization . . . 125
6.2.1 Overview . . . 125
6.2.2 Software Implementation . . . 125
6.2.3 Benchmark Setup . . . 125
6.2.4 Benchmark Results . . . 126
6.2.5 Performance Analysis . . . 126
6.2.6 Discussion . . . 129
6.2.7 Key Observations . . . 129
6.3 Matrix Multiplication . . . 129
6.3.1 Overview . . . 129
6.3.2 Software Implementation . . . 130
6.3.3 Benchmark Setup . . . 130
6.3.4 Benchmark Results . . . 130
6.3.5 Performance Analysis . . . 134
6.3.6 Discussion . . . 134
6.3.7 Key Observations . . . 135
6.4 Convolution . . . 135
6.4.1 Overview . . . 135
6.4.2 Software Implementation . . . 135
6.4.3 Benchmark Setup . . . 135
6.4.4 Benchmark Results . . . 135
6.4.5 Performance Analysis . . . 138

6.4.6 Discussion . . . 138
6.4.7 Key Observations . . . 138
6.5 Pooling . . . 138
6.5.1 Overview . . . 138
6.5.2 Software Implementation . . . 139
6.5.3 Benchmark Setup . . . 139
6.5.4 Benchmark Results . . . 140
6.5.5 Performance Analysis . . . 140
6.5.6 Discussion . . . 140
6.5.7 Key Observations . . . 141
6.6 Software Benchmark Summary . . . 141
6.6.1 Overall Speedup Comparison . . . 141
6.6.2 Functions Benefiting Most from Vectorization . . . 141
6.6.3 Computationally Intensive Operations . . . 141
6.6.4 Effectiveness of Software Optimizations . . . 143
6.6.5 Overall Conclusions . . . 143
6.6.6 Key Observations, Summary . . . 143

6.1 Nonlinear Activation Functions

Nonlinear activation functions are a core part of deep neural networks, enabling the learning of complex, non-linear relationships in data. Without these functions, a neural network of arbitrary depth would reduce to a linear transformation, thereby limiting its expressive capacity. For edge inference on a small core like the CV32E40P, the efficient implementation of activation functions is critical, as they are invoked at every layer of a network during both inference and feature extraction. This section presents the benchmarking results for four widely employed activation functions: Rectified Linear Unit (ReLU), Gaussian Error Linear Unit (GELU), Sigmoid, and Softmax. All implementations operate on fixed-point integer representations and leverage the RISC-V Vector Extension (RVV) to exploit data-level parallelism.

6.1.1 Rectified Linear Unit (ReLU)

Overview

The Rectified Linear Unit (ReLU) is the most widely deployed activation function in modern convolutional and fully connected neural networks. Defined as $f(x) = \max(0, x)$, ReLU introduces non-linearity with minimal computational overhead, clamping all negative activations to zero while preserving positive values. It has several practical advantages: it is computationally inexpensive, avoids the vanishing gradient problem that afflicts sigmoidal activations, and produces sparse activation maps that can improve generalization.
ReLU is applied after each convolutional or dense layer, typically following batch normalization. Its simplicity makes it an ideal candidate for vectorized implementation, as the operation reduces to a single element-wise maximum instruction with a scalar zero operand.

Software Implementation

The scalar implementation iterates over each element of the input array and applies a conditional assignment: if the element is positive, it is copied to the output; otherwise, zero is written. On the CV32E40P scalar core, this translates to a load, comparison, conditional store, and pointer-increment sequence per element.
The vectorized implementation exploits the RISC-V V extension (RVV) intrinsics with LMUL=2 (m2) and SEW=8 bits (int8 data type). The entire operation is reduced to a single vector maximum instruction, `vmax.vx`, which compares each lane of the loaded vector register against a scalar zero and produces the maximum. A strip-mining loop handles arbitrarily sized inputs: the body processes `vl_max` elements per iteration using full-width loads (`vle8.v`) and stores (`vse8.v`), while the tail handles any remaining elements by setting a reduced vector length. This approach ensures that no scalar fallback loop is required and that the vector unit operates at near-peak utilization throughout.

Benchmark Setup

Benchmarks were conducted on the CV32E40P core augmented with the VXT vector unit, configured with VLEN=128 bits. The data type is int8 (signed 8-bit integer), yielding a maximum vector length (VLMAX) of 32 elements per vector register group under LMUL=2. Input sizes range from 1,024 to 65,536 elements. Three input distribution scenarios were evaluated to characterize the sensitivity of the implementation to the positive-to-negative ratio of activations:

- **Scenario A (Post-Batch-Normalization):** 50% positive, 50% negative , the standard post-BN distribution.

- **Scenario B (No-BN Early Layers):** 70% positive, 30% negative , characteristic of early convolutional layers without batch normalization.
- **Scenario C (Dying ReLU / Worst Case):** 20% positive, 80% negative , simulates a pathological case in which most neurons are inactive.

Benchmark Results

Table 6.1: ReLU benchmark results, CV32E40P + VXT (VLEN=128, LMUL=2, SEW=8, int8).

Input Size	Scenario	Scalar Cycles	Vector Cycles	Speedup
1,024	A (Post-BN)	12,291	608	20.22×
4,096	A (Post-BN)	51,202	1,677	30.53×
4,096	B (No-BN Early)	52,021	1,677	31.02×
4,096	C (Dying ReLU)	49,973	1,677	29.80×
8,192	A (Post-BN)	102,402	3,341	30.65×
16,384	A (Post-BN)	204,802	6,669	30.71×
32,768	A (Post-BN)	409,602	13,325	30.74×
65,536	B (No-BN Early, int8)	832,309	26,637	31.25×

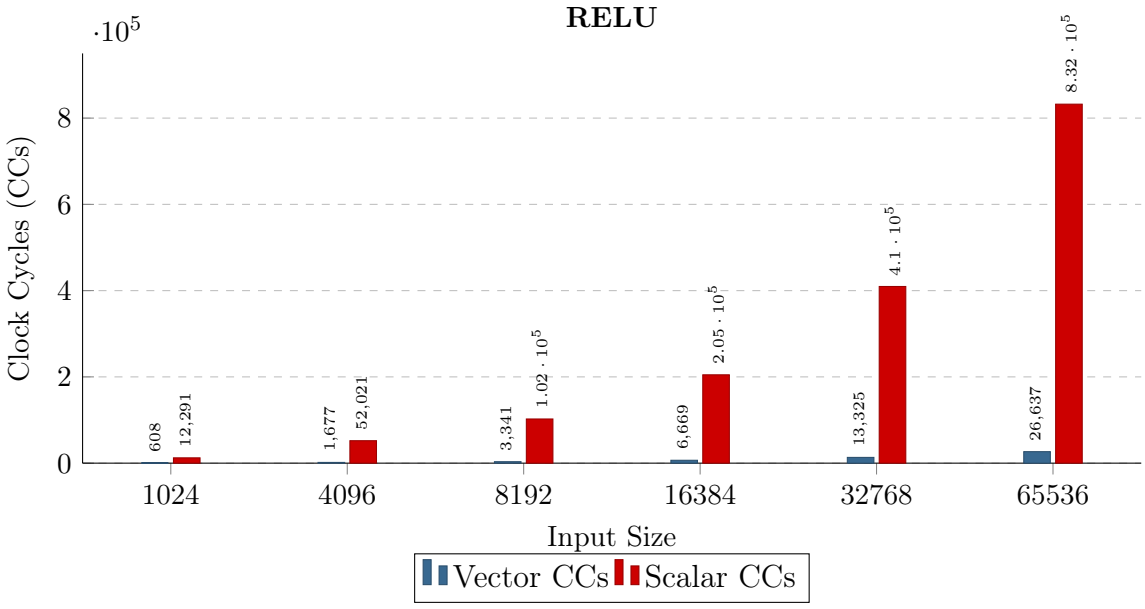

Figure 6.1: ReLU: Vector vs. Scalar clock cycles across input sizes.

Performance Analysis

The benchmark results demonstrate that the vectorized ReLU implementation achieves a sustained speedup of approximately 30–31× for input sizes of 4,096 elements and above, closely approaching the theoretical throughput gain predicted by the vector length under LMUL=2 (VLMAX = 32 elements). The near-linear relationship between scalar and vector cycle counts across all large input sizes confirms that both implementations are compute-bound, with no significant memory bandwidth bottleneck at these data sizes.

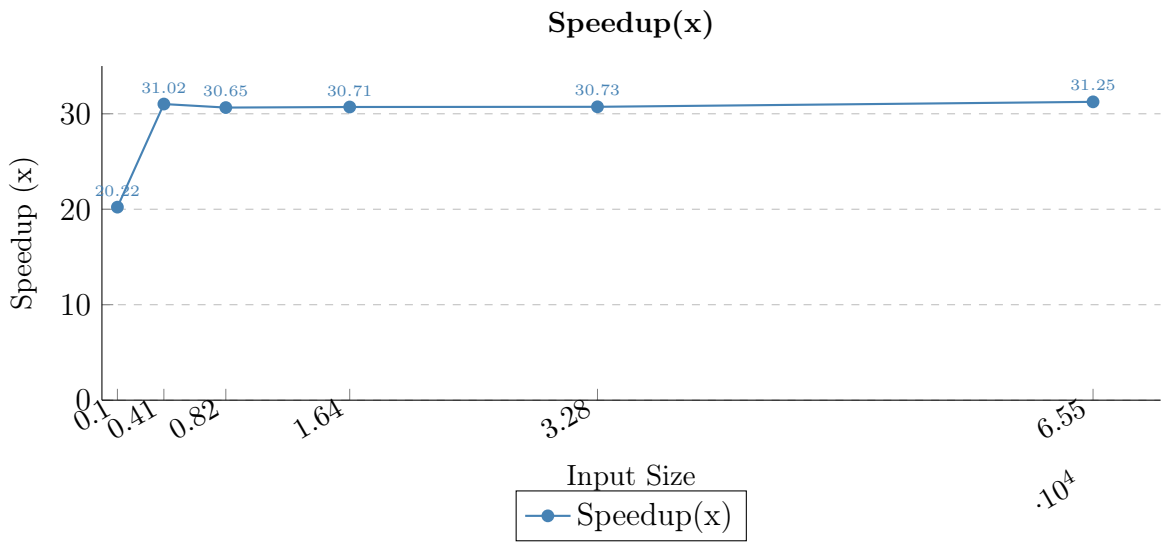

Figure 6.2: ReLU: Speedup (scalar/vector) as a function of input size.

At 1,024 elements, the observed speedup drops to 20.22×. This reduction is attributable to the proportionally higher weight of loop-control overhead, function-call prologue, and strip-mining tail handling relative to the total computation. As the input size increases, these fixed costs are amortized over a larger number of vector iterations, and the speedup converges toward the theoretical maximum.
The negligible variation in speedup across Scenarios A, B, and C at 4,096 elements (ranging from 29.80× to 31.02×) confirms that the vectorized implementation is entirely branch-free. The scalar implementation, by contrast, exhibits slight variation in cycle counts depending on the distribution of positive and negative values, suggesting that the branch predictor achieves high but not perfect prediction accuracy. The vector `vmax.vx` instruction processes all lanes unconditionally, eliminating data-dependent branch mispredictions entirely.
The near-constant vector cycle count of 1,677 cycles for all three 4,096-element Scenario A/B/C configurations confirms that the vector execution time is determined solely by the number of vector load/compute/store iterations, independent of the data content.

Discussion

ReLU achieves the highest vectorization speedup of all functions evaluated in this study, reaching up to 31.25×. This result is consistent with the function's computational profile: the operation consists of a single element-wise comparison with a scalar constant, which maps directly onto a single vector instruction with no inter-lane dependencies, no iterative convergence, and no lookup table accesses. The comparison demonstrates that for purely element-wise, branch-free operations on compact integer data types, the VXT vector unit can deliver speedups approaching the ratio of VLMAX to the scalar throughput of one element per clock.
The slight sub-linear speedup relative to VLMAX=32 is explained by the overhead of the strip-mining loop (one `setvl`, one branch, and two pointer updates per iteration) and the memory hierarchy latency, both of which add a small but measurable fixed cost per vector iteration.

Key Observations

- Sustained speedup of 30–31× achieved for $N \geq 4{,}096$, approaching the theoretical VLMAX=32 bound.

- The implementation is entirely branch-free; no data-dependent performance variation is observed in the vector path.
- Speedup is reduced to 20.22× at $N = 1{,}024$ due to loop-control and tail-handling overhead.
- All three distribution scenarios (A, B, C) yield identical vector cycle counts at the same input size.
- The scalar implementation exhibits minor data-dependent variation across scenarios, consistent with branch predictor sensitivity.

6.1.2 Gaussian Error Linear Unit (GELU)

Overview

The Gaussian Error Linear Unit (GELU) is an activation function defined as $\text{GELU}(x) = x \cdot \Phi(x)$, where $\Phi(x)$ denotes the cumulative distribution function of the standard normal distribution. Unlike ReLU, which produces a hard threshold at zero, GELU applies a smooth, probabilistic gating that weights each input by the probability that a Gaussian random variable exceeds it. GELU has become the de facto activation function in transformer-based architectures (including BERT, GPT, and ViT) due to its superior empirical performance compared to ReLU on language and vision tasks.
The direct computation of $\Phi(x)$ via the error function (erf) is computationally infeasible on integer-arithmetic processors. The implementation presented here employs a piecewise linear approximation (PLA) in a fixed-point Q3.12 format, enabling accurate evaluation of GELU without floating-point hardware.

Software Implementation

Both the scalar and vector implementations operate on int16 data in Q3.12 fixed-point format (one sign bit, three integer bits, twelve fractional bits). The GELU PLA decomposes the activation into five linear segments over the domain $|x| \in [0, 3.75)$, with elements beyond this range clamped: positive inputs are passed through unchanged (approaching the identity), while negative inputs are set to zero (approaching the asymptotic behavior of GELU). The correction term $C(|x|) = \text{GELU}(-|x|)$ is computed from the absolute value using the segment slopes and biases, and the final output is formed as $y = x + C(|x|)$ for $x \geq 0$, or $y = C(|x|)$ for $x < 0$.
The vectorized implementation uses RVV intrinsics with LMUL=2 and SEW=16 (int16), yielding VLMAX=16 elements per vector register group. All five segment comparisons are computed simultaneously using vector mask operations (`vmslt.vx`), and the segment values are selected via `vmerge` instructions in a nested override pattern. The `vmulh.vx` instruction is used for fixed-point multiplication, avoiding overflow. This branchless, mask-based evaluation processes all lanes in parallel with a constant number of vector instructions per strip-mining iteration.

Benchmark Setup

Benchmarks were conducted under Profile A (Attention), which models the logit distributions encountered in transformer self-attention layers. Input values are drawn from a non-uniform distribution across seven amplitude ranges, with the majority of values concentrated in the mid-range segments (S5–S7) where the GELU approximation is most active. Input sizes range from 512 to 16,384 elements (int16). The benchmark measures total clock cycles from function entry to return.

Benchmark Results

Table 6.2: GELU benchmark results, Profile A (Attention), Q3.12 int16, LMUL=2, SEW=16.

Input Size (N)	Scalar Cycles	Vector Cycles	Speedup
512	15,015	2,955	5.08×
1,024	30,091	5,867	5.13×
2,048	58,455	11,819	4.95×
3,072	87,754	17,707	4.96×
4,096	116,868	23,596	4.95×
8,192	233,402	47,148	4.95×
16,384	466,389	94,252	4.95×

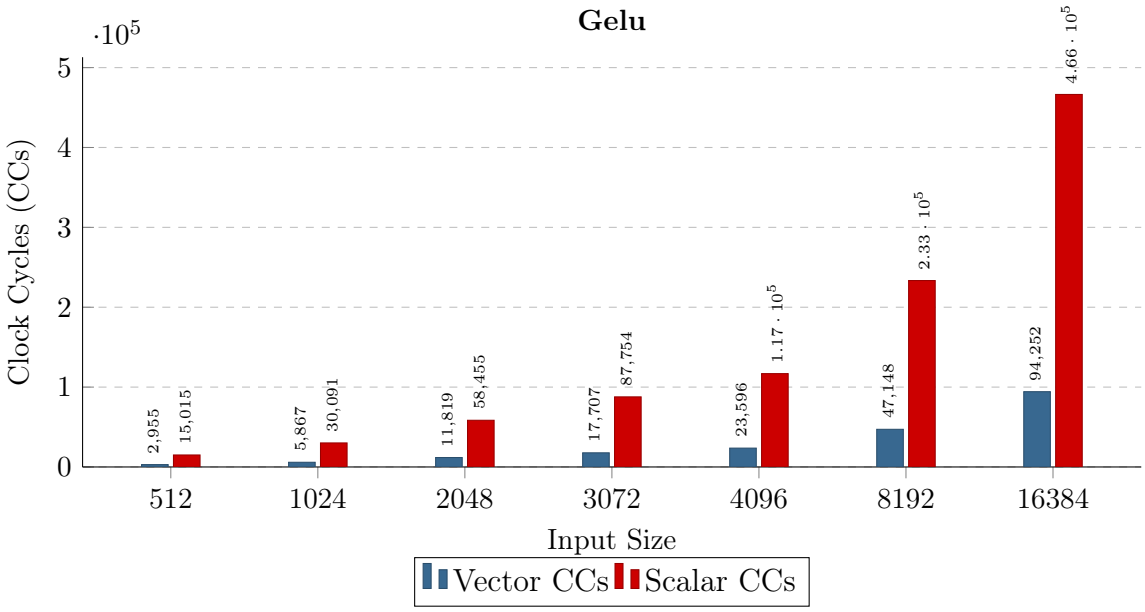

Figure 6.3: GELU: Vector vs. Scalar clock cycles across input sizes (Profile A, Attention).

Performance Analysis

The GELU vectorized implementation achieves a stable speedup of approximately 4.95–5.13× across all evaluated input sizes. This speedup is substantially lower than that achieved by ReLU, which is expected given the significantly higher computational complexity of the five-segment PLA evaluation. Each strip-mining iteration requires: one vector load, one absolute-value computation (`vmax` + `vrsub`), five comparison masks (`vmslt`), five multiply-high operations (`vmulh`), five add operations, five merge operations (`vmerge`), one sign-based select (`vmerge`), and one vector store, a total of approximately 22 vector instructions per iteration, compared to three for ReLU.

Despite the higher instruction count, the vector implementation still achieves a meaningful speedup because all instructions operate across VLMAX=16 lanes simultaneously. The scalar equivalent must execute the same logical sequence on each of the 16 elements individually, with additional branch overhead for the segment selection logic. The very stable speedup across all input sizes (4.95–4.96× for $N \geq 2{,}048$) indicates that the implementation is fully throughput-limited with no cache miss penalty, and that the strip-mining overhead is negligible at these sizes.

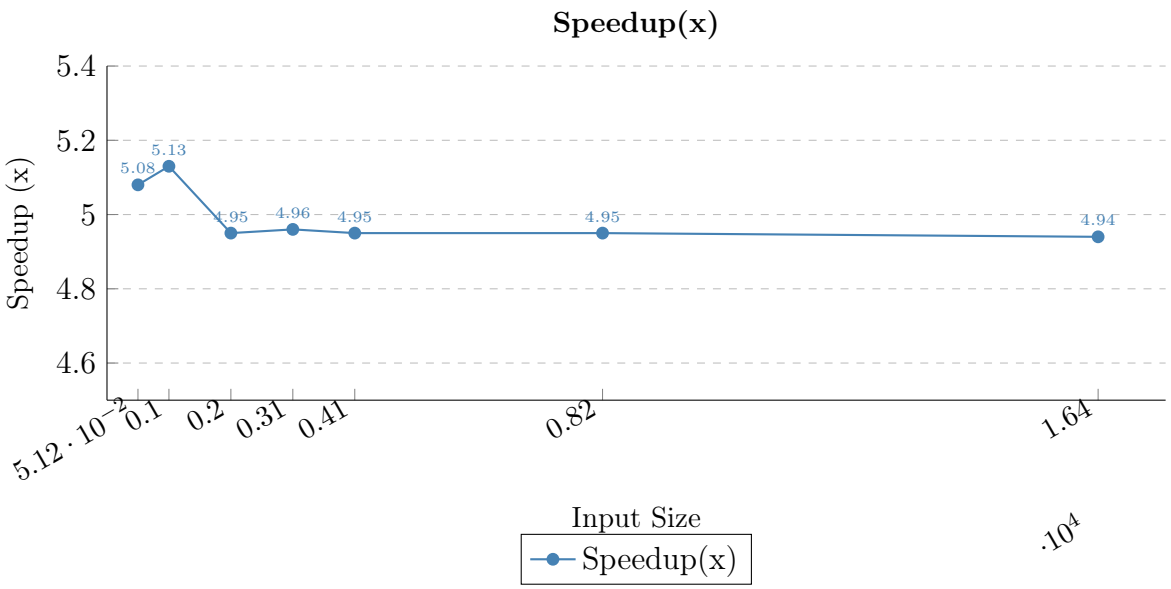

Figure 6.4: GELU: Speedup as a function of input size (Profile A, Attention).

The slightly higher speedup at $N = 512$ (5.08×) and $N = 1{,}024$ (5.13×) may reflect cache warm-up effects or minor variations in scalar branch prediction at smaller sizes.

Discussion

GELU achieves a consistent speedup of approximately 4.95× across a wide range of input sizes, confirming that the PLA-based vectorized evaluation scales linearly with input length. The comparison demonstrates that the five-segment approximation introduces roughly a 6× computational overhead relative to ReLU on the scalar path; however, the vector implementation effectively amortizes this overhead by processing 16 lanes in parallel with branchless mask operations. The near-constant speedup across all sizes indicates excellent scalability and confirms that the implementation is not memory-bandwidth-limited for the tested configurations.

Key Observations

- Sustained speedup of approximately 4.95× across all input sizes from 2,048 to 16,384 elements.
- GELU is significantly more complex than ReLU due to the five-segment PLA, but the vectorized implementation handles all segments with branchless mask operations.
- The speedup is fundamentally limited by the compute intensity of the PLA (approximately 22 vector instructions per iteration) relative to the VLMAX=16 parallelism factor.
- The scalar implementation incurs additional branch overhead for segment selection, partially explaining the speedup advantage.
- The fixed-point Q3.12 representation enables accurate GELU approximation without floating-point hardware.

6.1.3 Sigmoid

Overview

The sigmoid function, defined as $\sigma(x) = \frac{1}{1+e^{-x}}$, maps any real-valued input to the range $(0, 1)$, making it well-suited as an output activation for binary classification tasks and as a gating mechanism in recurrent architectures such as LSTMs and GRUs. In transformer-based models, sigmoid gating appears in attention variants and gated linear units (GLUs). As with GELU, direct evaluation of the exponential function is not feasible on integer processors, necessitating a PLA-based approximation.
The implemented approximation covers the symmetric sigmoid response over the domain $|x| \in [0, 5.5)$ using five linear segments, with values beyond this range clamped to 0.0 or 1.0. The representation uses Q3.12 fixed-point int16 arithmetic.

Software Implementation

The scalar sigmoid implementation evaluates the piecewise linear approximation using sequential conditional comparisons and multiplications, with one branch per segment boundary. The vector implementation mirrors the scalar logic but applies all comparisons and linear evaluations simultaneously across VLMAX=16 lanes using mask registers. The implementation uses `vmslt.vx` to compute all four segment boundary masks, `vmulh.vx` for fixed-point slope multiplication, `vadd.vx` for the bias term, and `vmerge` to select the appropriate segment output for each lane. An additional pass handles symmetry by computing the absolute value and applying the sign to reflect the sigmoid's odd symmetry about 0.5.
The default (clamped) value of `SCALE` = 4096 (representing 1.0 in Q3.12) is broadcast to all lanes at the start, and segment values progressively overwrite this default for lanes falling within each segment boundary, from outermost to innermost, using masked merges.

Benchmark Setup

Benchmarks were conducted under Profile B (Classification), which reflects the logit distributions encountered in classification networks (e.g., final-layer logits in MobileNet). Input values are drawn from a distribution concentrated at higher absolute magnitudes, where the sigmoid response is in its saturation region (segments S3 and S4), with a significant fraction of inputs clamped. Input sizes range from 128 to 8,192 elements (int16, Q3.12).

Benchmark Results

Table 6.3: Sigmoid benchmark results, Profile B (Classification), Q3.12 int16, LMUL=2, SEW=16.

Input Size (N)	Scalar Cycles	Vector Cycles	Speedup
128	4,795	805	5.96×
256	9,546	1,557	6.13×
512	19,085	3,061	6.23×
1,024	38,163	6,069	6.29×
2,048	78,934	11,960	6.60×
4,096	157,830	23,864	6.61×
8,192	315,660	47,672	6.62×

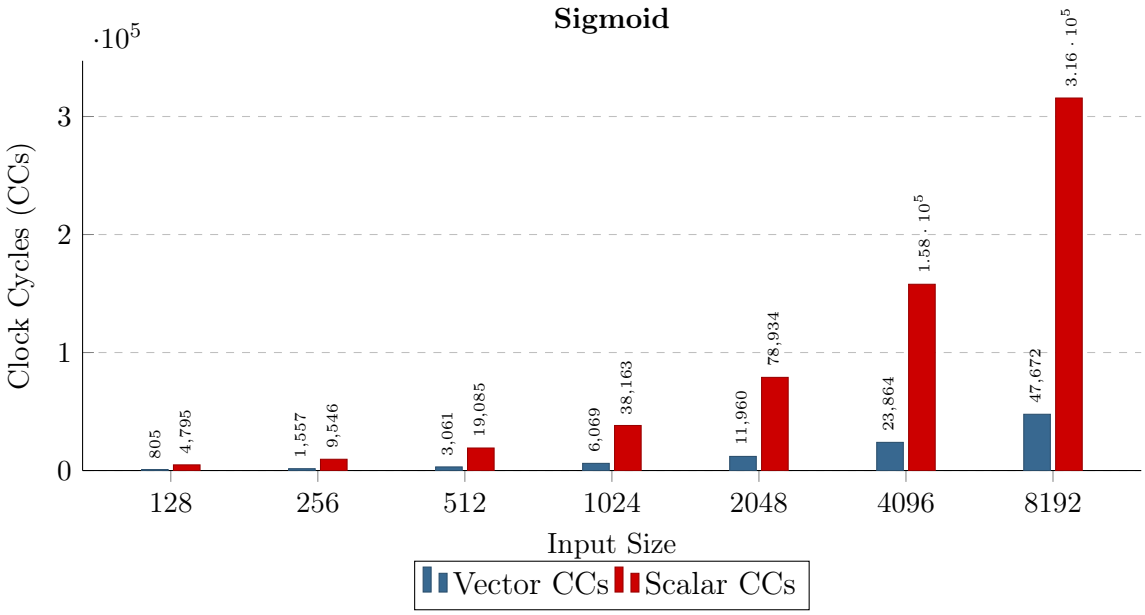

Figure 6.5: Sigmoid: Vector vs. Scalar clock cycles across input sizes (Profile B, Classification).

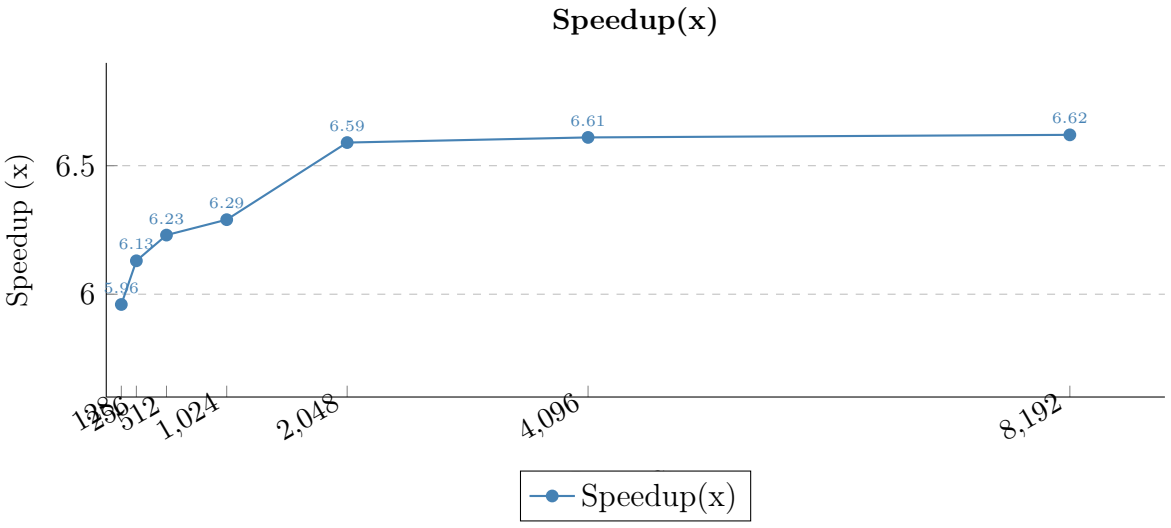

Figure 6.6: Sigmoid: Speedup as a function of input size (Profile B, Classification).

Performance Analysis

The sigmoid implementation achieves a speedup that increases from 5.96× at $N = 128$ to a stable 6.61–6.62× for $N \geq 4{,}096$. The monotonically increasing speedup with input size reflects the amortization of fixed per-call overhead (function prologue, `setvlmax` call, and the first `vl` setting) over a larger number of strip-mining iterations. At $N = 128$, this overhead constitutes a larger fraction of total execution time, depressing the speedup. At $N = 4{,}096$ and above, the speedup stabilizes, indicating that both the scalar and vector implementations are operating in their asymptotic throughput regimes.
The sigmoid speedup of 6.61× is higher than the GELU speedup of 4.95×, despite both functions using the same number of segments (five for GELU, five for sigmoid). The difference arises from two factors: first, the sigmoid's PLA is slightly less instruction-intensive than GELU because the symmetry handling for sigmoid is simpler (the sigmoid is odd-symmetric about 0.5, requiring only a single reflect operation), whereas GELU requires computing $x + C(|x|)$ for positive x and $C(|x|)$ for negative x; second, the scalar sigmoid implementation may incur slightly higher branch misprediction rates under Profile B due to the concentration of inputs in the saturation segments, increasing the relative cost of scalar execution.

Discussion

The sigmoid vector implementation achieves a consistent speedup range of 5.96–6.62×, with the characteristic increase from smaller to The speedup grows with input size as per-call overhead amortization. The comparison demonstrates that the PLA-based approach is effective for sigmoid evaluation in the classification profile, where the majority of inputs are in the saturation region and the clamped default value is selected for a significant fraction of lanes, an operation that is free in the vector implementation (the default is initialized once via `vmv.v.x`) but incurs a branch in the scalar path.

Key Observations

- Speedup increases monotonically from 5.96× ($N = 128$) to 6.62× ($N = 8{,}192$) as per-call overhead is amortized.
- Sigmoid achieves a higher speedup than GELU despite equivalent segment count, owing to simpler symmetry handling and favorable scalar branch behavior under Profile B.
- The stable asymptotic speedup of 6.61–6.62× indicates that the implementation is compute-bound with no memory bandwidth limitation.
- The clamped default case (saturation region) incurs no additional cost in the vector path, as the default vector is initialized once via a broadcast.

6.1.4 Softmax

Overview

The softmax function normalizes a vector of real-valued logits into a probability distribution:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}.$$

It is ubiquitous as the final activation in multi-class classifiers and as the attention weighting function in transformer self-attention layers. Softmax is computationally demanding relative to

element-wise activations because it involves two sequential reduction passes over the input (one for max subtraction and one for the partition function $\sum e$), in addition to the element-wise exponential evaluations.
The implementation adopts the numerically stable formulation: first, the maximum logit is subtracted from all inputs, and then $\exp(-|x - \max|)$ is computed element-wise using a seven-segment PLA in Q3.12 fixed-point format. The partition function is accumulated via a vector reduction, and the normalized outputs are computed using integer division.

Software Implementation

The exponential approximation employs a seven-segment piecewise linear function over the domain $[0, 8.0)$ in Q3.12 fixed-point (`EXP_ZERO_THRESH` = 32727, representing approximately 7.99). Inputs beyond this threshold are clamped to zero ($\exp(-\infty) \approx 0$). The steeper segments (S1–S3, covering the rapid decay region near $x = 0$) use an additional left shift (`vsll` by 2) after the `vmulh` operation to recover precision, while the flatter segments (S4–S7) use standard `vmulh` encoding. This dual encoding strategy is implemented identically in both scalar and vector paths to ensure numerical consistency.
Two input distribution profiles were evaluated: Profile A (Attention), reflecting transformer self-attention logit distributions where values span the full dynamic range; and Profile B (Classification), modeling classifier output logits with a heavy concentration of inputs in the large-magnitude clamped region (43% of inputs are zero-clamped under Profile B).

Benchmark Setup

Softmax benchmarks were conducted over input sizes representative of both attention ($N = 128$ to 2,048) and classification ($N = 10$ to 32,000) workloads. Profile A is used for attention sizes; Profile B for classification sizes. All inputs are int16 in Q3.12 format. The metric includes the full softmax computation: max-subtraction pass, exponential evaluation pass, reduction sum, and normalization.

Benchmark Results

Profile A, Attention ($N = 128$ to 2,048):

Table 6.4: Softmax benchmark results, Profile A (Attention), Q3.12 int16, LMUL=2, SEW=16.

Input Size (N)	Scalar Cycles	Vector Cycles	Speedup
128	11,364	1,497	7.59×
256	22,647	2,800	8.09×
512	45,182	5,407	8.36×
1,024	90,260	10,622	8.50×
2,048	176,101	21,807	8.08×

Profile B, Classification ($N = 10$ to 32,000):

Performance Analysis

The softmax benchmarks reveal several noteworthy trends. Under Profile A (Attention), the speedup increases from 7.59× at $N = 128$ to a peak of 8.50× at $N = 1{,}024$, then decreases slightly to 8.08× at $N = 2{,}048$. This non-monotonic behavior is explained by the interaction between the vector reduction operation (which has fixed overhead regardless of input size)

Table 6.5: Softmax benchmark results, Profile B (Classification), Q3.12 int16, LMUL=2, SEW=16.

Input Size (N)	**Scalar Cycles**	**Vector Cycles**	**Speedup**
10	940	370	2.54×
80	6,346	1,010	6.28×
1,000	76,280	10,825	7.05×
32,000	2,437,809	338,172	7.21×

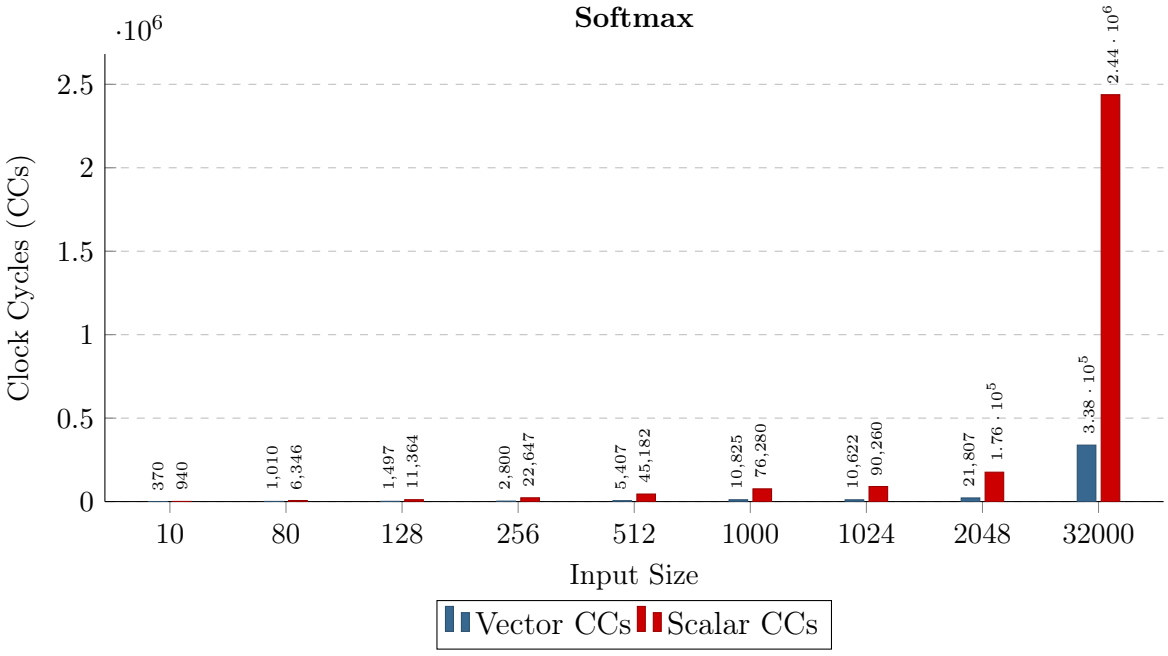

Figure 6.7: Softmax: Vector vs. Scalar clock cycles across all input sizes (Profiles A and B combined).

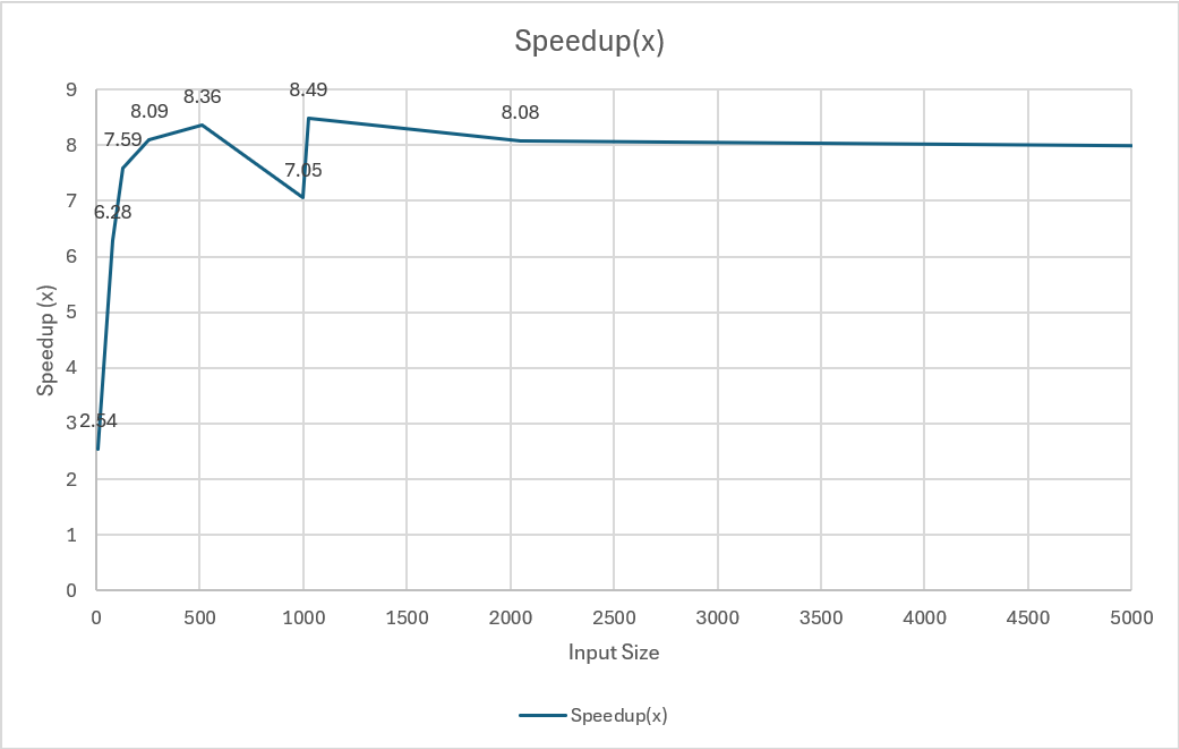

Figure 6.8: Softmax: Speedup as a function of input size (combined Profiles A and B).

and the element-wise exponential pass. For moderate sizes such as $N = 1{,}024$, the ratio of productive computation to reduction overhead is maximized, yielding the peak speedup. At $N = 2{,}048$, the absolute overhead of the reduction grows slightly relative to the benefit, marginally reducing the speedup.

Under Profile B (Classification), the speedup at $N = 10$ (2.54×) is drastically lower than at larger sizes. This is expected: for very small inputs, the scalar implementation completes in only 940 cycles (primarily function-call and `setvl` overhead) and the vector unit's per-call setup cost dominates. At $N = 80$ (6.28×) and beyond, the speedup recovers to values consistent with the compute-bound regime. At $N = 32{,}000$, the speedup of 7.21× reflects the large fraction of zero-clamped inputs under Profile B (43%), which reduces the effective computation per element in both scalar and vector paths, but the vector implementation benefits more due to the branchless handling of the clamped default.

The softmax achieves a higher speedup than GELU (approximately 8.5× vs. 4.95×) despite the additional reduction passes. This is primarily because the exponential PLA has seven segments with a partially precombined evaluation strategy, and the scalar softmax must execute sequential scalar reductions that are particularly slow compared to the vector reduction primitives.

Discussion

The softmax function achieves the highest speedup among all PLA-based activations in this study, reaching up to 8.50× for the attention profile. The comparison demonstrates that the combination of an efficient seven-segment exp approximation with vectorized reduction operations provides a substantially more favorable scalar-to-vector ratio than simpler activations evaluated on fewer segments. The poor performance at $N = 10$ is a well-known limitation of vector processing for very small inputs and does not diminish the effectiveness of the implementation for practical neural network workloads, where softmax dimensions typically range from 128 to tens of thousands.

Key Observations

- Peak speedup of 8.50× achieved at $N = 1{,}024$ under Profile A (Attention).
- Profile B speedup at $N = 10$ is only 2.54× due to fixed per-call overhead dominating the computation; this recovers to 7.21× at $N = 32{,}000$.
- Softmax achieves higher speedup than GELU, attributed to the efficiency of vectorized reductions and the relatively higher cost of scalar sequential reduction.
- The seven-segment exponential PLA uses a dual-encoding strategy (`vsll` for steep segments, standard `vmulh` for flat segments) to maintain numerical fidelity across the full domain.
- The zero-clamped region (Profile B) incurs no extra cost in the vector path due to the branchless default initialization.

6.2 Requantization

6.2.1 Overview

Requantization is the process of converting an intermediate integer accumulation from a higher-precision representation to a lower-precision format, typically following a convolution or matrix multiplication operation. In integer-quantized neural network inference pipelines, the multiply-accumulate operations produce outputs in a wider integer format (e.g., int32 or int16) to prevent overflow. Before the result is passed to the next layer, it must be scaled, shifted, and saturated back into the target precision of that layer (e.g., int16 or int8). This operation is computationally simple, involving a right shift, saturation clamp, and optional bias addition, but is invoked on every output element after every linear layer, making its efficiency critical to overall inference throughput.
Three requantization variants were implemented and benchmarked: int32 → int16, int32 → int8, and int16 → int8. Each variant reduces the bit-width of the data by a factor of two or four, with the primary operations being a saturating narrowing right-shift followed by a type-cast.

6.2.2 Software Implementation

The scalar implementations perform a right shift by the quantization scale factor, followed by a saturation clamp to the target type's representable range, on each element individually. The vectorized implementations exploit the RVV narrowing saturating right shift instructions (`vnsra` / `vnsrl`), which perform the shift and saturation in a single vector operation. For int32 → int16, the input is loaded as `vint32m4` and narrowed to `vint16m2` using `vnsra.wi`. For int32 → int8, a two-stage narrowing is applied (int32 → int16 → int8). For int16 → int8, a single `vnsra.wi` with SEW=16 and LMUL=2 produces `vint8m1` output. The use of saturating narrowing instructions eliminates the need for explicit clamp comparisons and ensures that the output values remain within the target type's range without additional instructions.

6.2.3 Benchmark Setup

Each requantization variant was benchmarked across four input sizes: 128, 512, 1,024, and 4,096 elements. The benchmark measures the total clock cycles required to perform the narrowing operation on the full input array, including all loads and stores. All variants use LMUL=2

for the output register group, with the input LMUL adjusted accordingly (LMUL=4 for int32 inputs).

6.2.4 Benchmark Results

Table 6.6: int32 → int16 requantization benchmark results (LMUL=2, SEW=16 output).

Input Size	Scalar Cycles	Vector Cycles	Speedup
128	2,571	265	9.70×
512	10,251	989	10.37×
1,024	20,491	1,965	10.43×
4,096	81,931	7,693	10.65×

Table 6.7: int32 → int8 requantization benchmark results (LMUL=2, SEW=8 output).

Input Size	Scalar Cycles	Vector Cycles	Speedup
128	5,287	406	13.02×
512	21,031	1,492	14.10×
1,024	42,023	2,940	14.29×
4,096	167,975	11,628	14.45×

Table 6.8: int16 → int8 requantization benchmark results (LMUL=2, SEW=8 output).

Input Size	Scalar Cycles	Vector Cycles	Speedup
128	2,698	135	19.99×
512	10,762	501	21.48×
1,024	21,514	989	21.75×
4,096	81,931	3,853	21.26×

6.2.5 Performance Analysis

The requantization benchmarks reveal a strong correlation between the precision reduction ratio and the observed speedup: int16 → int8 achieves the highest speedup (up to 21.75×), followed by int32 → int8 (up to 14.45×), and then int32 → int16 (up to 10.65×). This hierarchy is explained by the ratio of memory bandwidth to computation: the int16 → int8 conversion reduces the number of bytes per element from 2 to 1, allowing twice as many output elements to be stored per vector store instruction compared to int32 → int16. Additionally, the vector narrowing instruction for int16 → int8 under LMUL=2 loads 16 int16 elements and produces 16 int8 elements per vector group, effectively doubling the useful throughput per store compared to int32 → int16 which processes 8 int32 elements into 8 int16 elements.
For int32 → int8, the two-stage narrowing (int32 → int16 → int8) requires two narrowing passes, explaining the intermediate speedup. Despite the extra pass, the overall throughput benefit remains high because the scalar implementation also requires two explicit range checks and casts. The narrowing saturating right shift instruction (`vnsra`) subsumes both the shift

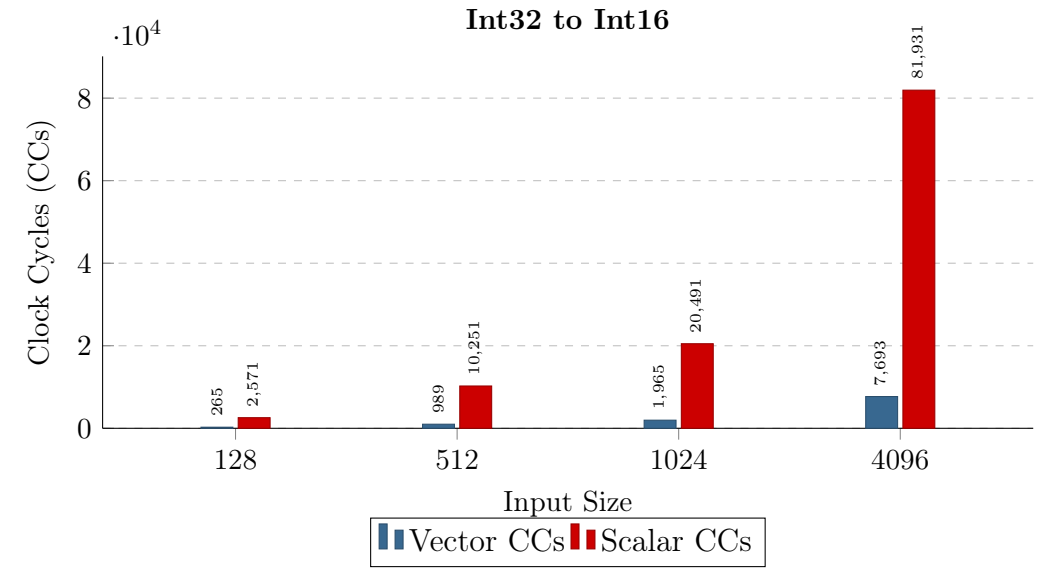

Figure 6.9: int32 → int16 Requantization: Vector vs. Scalar clock cycles.

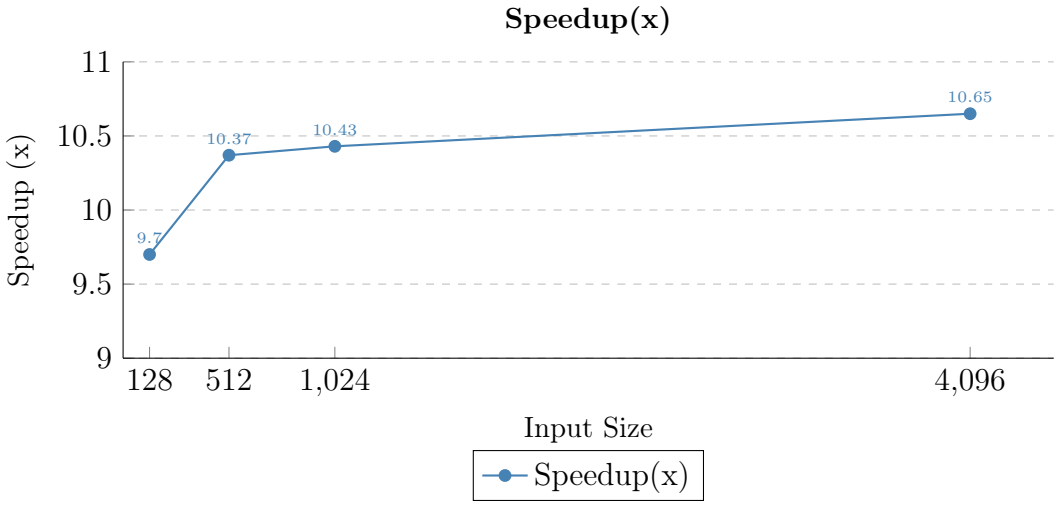

Figure 6.10: int32 → int16 Requantization: Speedup as a function of input size.

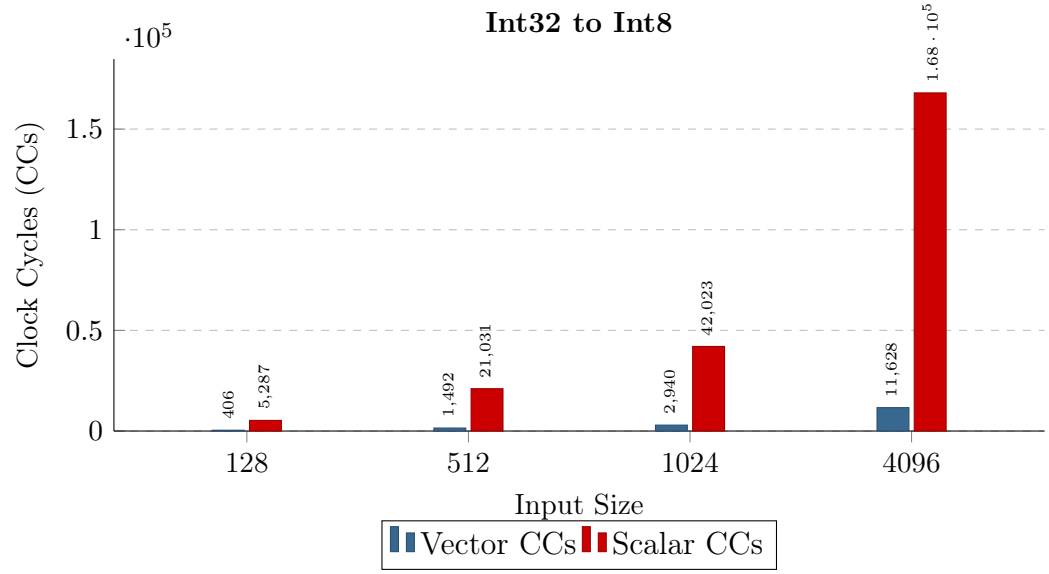

Figure 6.11: int32 → int8 Requantization: Vector vs. Scalar clock cycles.

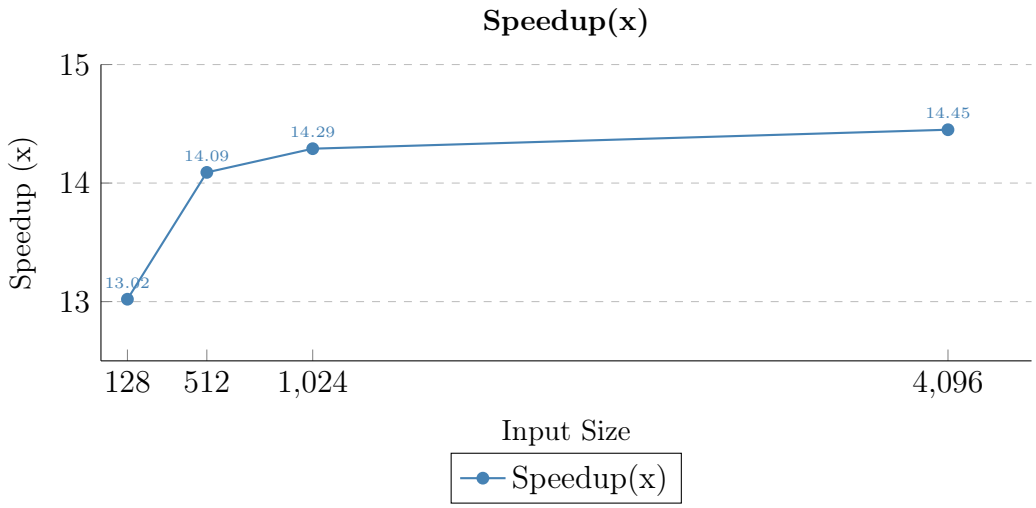

Figure 6.12: int32 → int8 Requantization: Speedup as a function of input size.

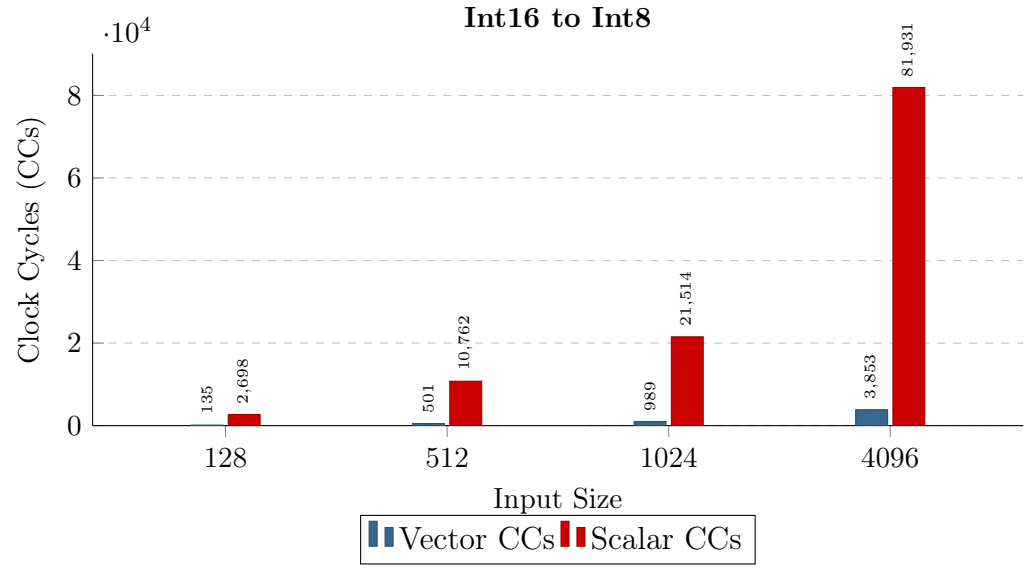

Figure 6.13: int16 → int8 Requantization: Vector vs. Scalar clock cycles.

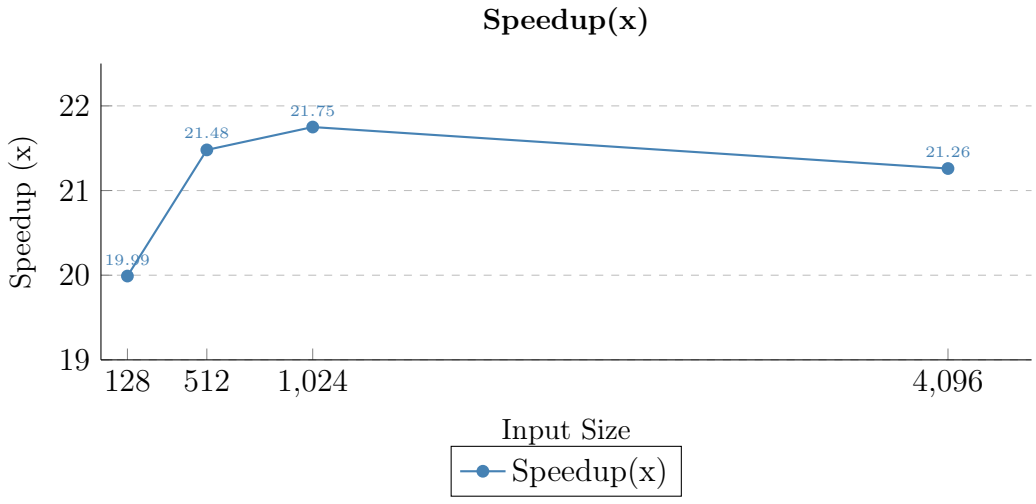

Figure 6.14: int16 → int8 Requantization: Speedup as a function of input size.

and the saturation in a single operation, delivering significantly higher throughput than the equivalent scalar sequence of shift, compare, and clamp.
The speedup increases monotonically with input size for all three variants, reflecting the amortization of per-call overhead. The exceptionally high speedup of int16 $\rightarrow$ int8 (21.75$\times$ at N = 1,024) places it among the highest-speedup implementations in this entire benchmark suite.

6.2.6 Discussion

Requantization operations benefit substantially from vectorization, with the int16 $\rightarrow$ int8 variant achieving the second-highest speedup observed across all benchmarked functions (exceeded only by ReLU). The comparison demonstrates that the RVV narrowing saturating instructions are particularly well-matched to requantization workloads: they eliminate explicit saturation comparisons, perform the right shift and type narrowing in a single instruction, and operate on twice as many logical elements per vector width compared to the widened input type. The high speedups across all variants confirm that requantization is a primary target for vectorization in quantized neural network inference pipelines.

6.2.7 Key Observations

- int16 $\rightarrow$ int8 achieves the highest speedup (up to 21.75$\times$), benefiting from a 2:1 element packing ratio under LMUL=2.
- int32 $\rightarrow$ int8 achieves up to 14.45$\times$, reflecting the two-stage narrowing cost partially offset by the elimination of explicit clamp instructions.
- int32 $\rightarrow$ int16 achieves up to 10.65$\times$, consistent with an 8-element-per-iteration throughput under LMUL=2 with SEW=32.
- Speedup increases with input size for all variants, confirming that the implementations are compute-bound in the asymptotic regime.
- The `vnsra.wi` instruction is the key enabler: it performs shift and saturation in a single operation, eliminating the multi-instruction scalar sequence.

6.3 Matrix Multiplication

6.3.1 Overview

Matrix multiplication is the foundational operation underlying fully connected layers, attention projections, and feed-forward network (FFN) sub-layers in transformer architectures. Given matrices $\mathbf{A}$ of dimension $M \times K$ and $\mathbf{B}$ of dimension $K \times N$, the product $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ has dimension $M \times N$, with each element $C[i][j]$ formed as the inner product of the i-th row of $\mathbf{A}$ and the j-th column of $\mathbf{B}$. The computational complexity is $O(M \cdot K \cdot N)$ multiply–accumulate (MAC) operations, which makes matrix multiplication the dominant cost in transformer inference for all but the smallest models. The kernel benchmarked here targets integer arithmetic with int8 inputs and int32 accumulation, and exploits data-level parallelism along the output column dimension using the RISC-V Vector Extension (RVV).

6.3.2 Software Implementation

The scalar implementation follows the standard triple-nested loop (i, j, k) with int32 accumulation, computing one element of **C** per scalar multiply–accumulate in the innermost loop. The vectorized implementation parallelizes over the output column dimension (the j-loop): for each row i and accumulation index k, the scalar contribution $A[i][k]$ is broadcast across all lanes, and a vector of $B[k][j : j + vl]$ elements is loaded and multiplied. The partial products are accumulated into a wide vector accumulator (int32) across all k iterations, and after the accumulation for row i is complete the result vector is written to the corresponding row of **C**. This arrangement maximizes reuse of the broadcast scalar $A[i][k]$ across all lanes, ensuring that the operand fetched from the scalar register file is amortized over an entire vector of **B**-row elements.
Two of the three benchmark groups additionally apply a bias term to the accumulated result prior to write-back. The benchmark source labels this group as employing an "optimization", but the nature of that optimization is not described in the available data and is therefore not characterized here. Whether both operands **A** and **B** are int8 is implied by the `in8` datatype label; the source does not separately specify per-operand datatypes. The first two groups produce int16 output, while the third group produces int32 output.

6.3.3 Benchmark Setup

Three configuration groups were benchmarked on the CV32E40P core augmented with the VXT vector unit:

- **Group 1**, int8 → int16, multiplication without bias addition (three configurations).
- **Group 2**, int8 → int16, multiplication with bias addition, labeled "optimization" (four configurations, all with output shape $\mathbf{C}[32 \times 10]$).
- **Group 3**, int8 → int32, multiplication with bias addition (three large configurations with $M = 1$).

Dimensions are reported as $\mathbf{A}[M \times K]$, $\mathbf{B}[K \times N]$, $\mathbf{C}[M \times N]$. One Group 1 configuration is annotated in the source as a "small transformer FFN." The metric is total clock cycles for the complete multiplication (and bias addition where applicable).

6.3.4 Benchmark Results

Group 1, int8 → int16, without bias addition:

Table 6.9: Matrix multiplication, int8→int16, no bias (CV32E40P + VXT).

Configuration ($\mathbf{A} \cdot \mathbf{B} \to \mathbf{C}$)	Scalar Cycles	Vector Cycles	Speedup
$A[32 \times 64] \cdot B[64 \times 8] \to C[32 \times 8]$	165,900	24,495	6.7728108×
$A[64 \times 64] \cdot B[64 \times 16] \to C[64 \times 16]$	597,644	98,767	6.051049×
$A[8 \times 128] \cdot B[128 \times 512] \to C[8 \times 512]$ (FFN)	5,271,628	684,087	7.706078×

Group 2, int8 → int16, with bias addition ("optimization"):
Group 3, int8 → int32, with bias addition:

Table 6.10: Matrix multiplication, int8→int16, with bias addition.

Configuration ($\mathbf{A} \cdot \mathbf{B} \rightarrow \mathbf{C}$)	**Scalar Cycles**	**Vector Cycles**	**Speedup**
$A[32 \times 32] \cdot B[32 \times 10] \rightarrow C[32 \times 10]$	107,861	23,006	4.688×
$A[32 \times 64] \cdot B[64 \times 10] \rightarrow C[32 \times 10]$	190,071	30,430	6.246×
$A[32 \times 128] \cdot B[128 \times 10] \rightarrow C[32 \times 10]$	415,061	45,535	9.115×
$A[32 \times 100] \cdot B[100 \times 10] \rightarrow C[32 \times 10]$	293,751	55,902	5.255×

Table 6.11: Matrix multiplication, int8→int32, with bias addition.

Configuration ($\mathbf{A} \cdot \mathbf{B} \rightarrow \mathbf{C}$)	**Scalar Cycles**	**Vector Cycles**	**Speedup**
$A[1 \times 4096] \cdot B[4096 \times 1000] \rightarrow C[1 \times 1000]$	36,888,042	7,738,214	4.766997×
$A[1 \times 4096] \cdot B[4096 \times 4096] \rightarrow C[1 \times 4096]$	167,870,504	32,744,536	5.126672×
$A[1 \times 9216] \cdot B[9216 \times 4096] \rightarrow C[1 \times 4096]$	377,585,704	73,376,856	5.145841×

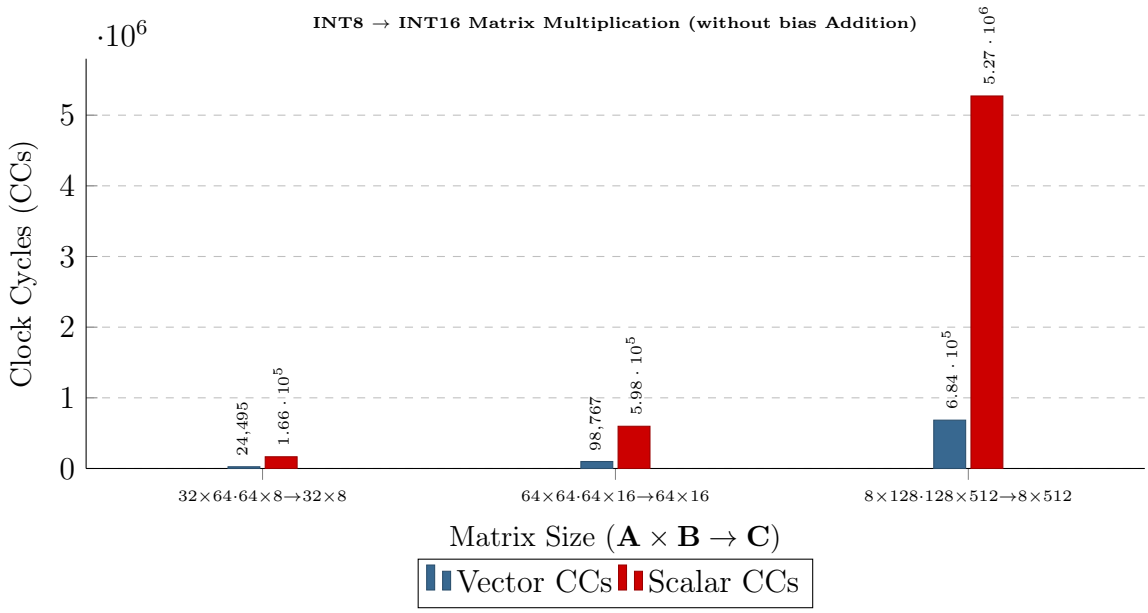

Figure 6.15: INT8→INT16 Matrix Multiplication (without bias): Vector vs. Scalar clock cycles.

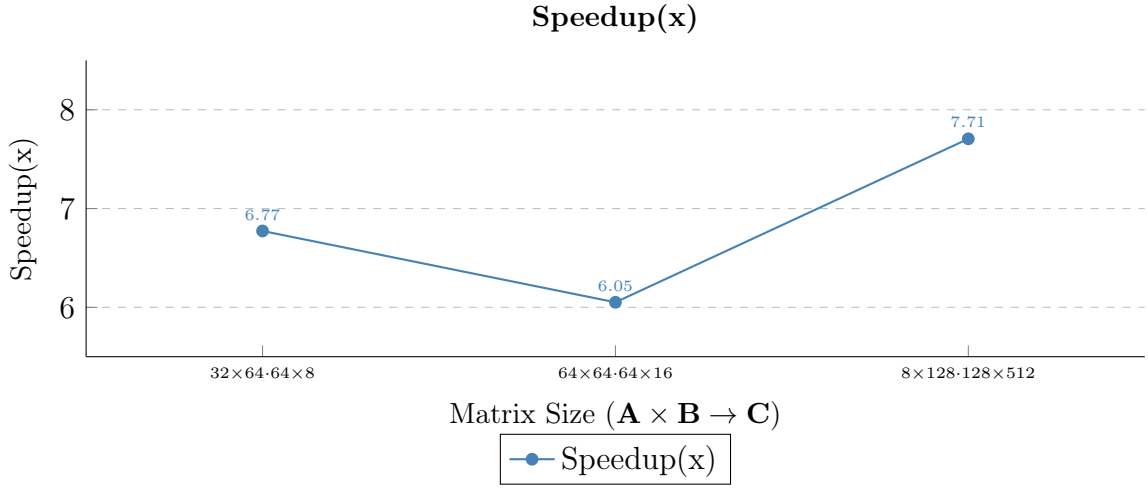

Figure 6.16: INT8→INT16 Matrix Multiplication (without bias): Speedup per configuration.

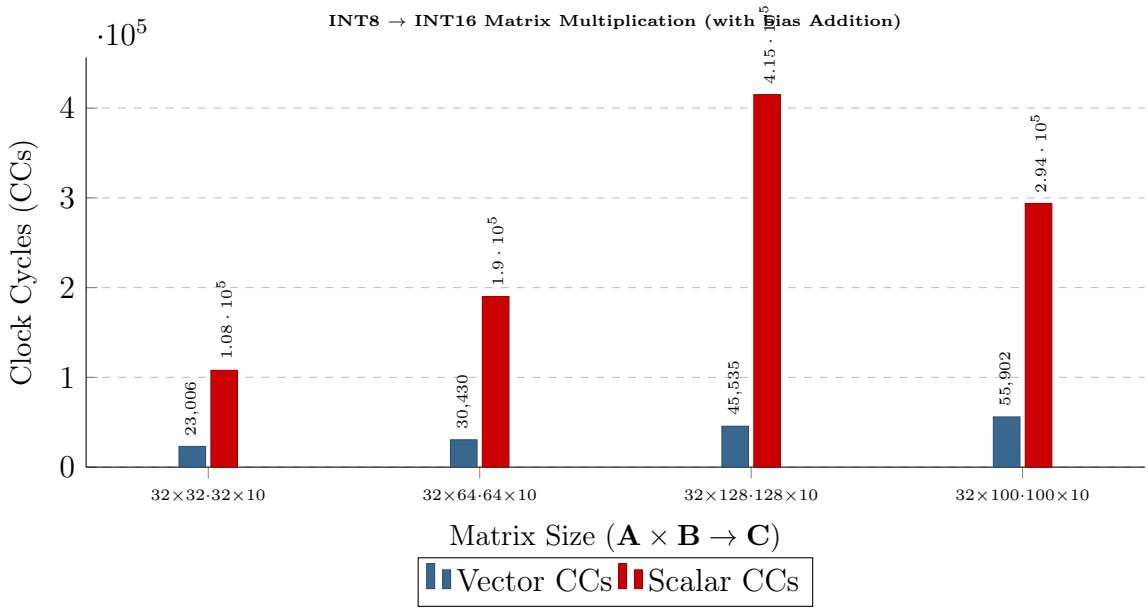

Figure 6.17: INT8→INT16 Matrix Multiplication (with bias): Vector vs. Scalar clock cycles.

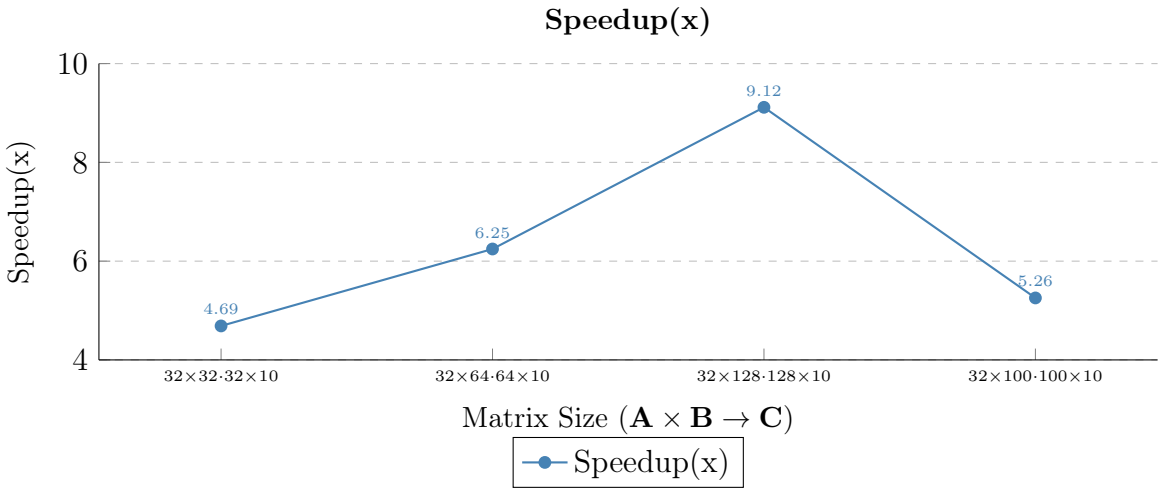

Figure 6.18: INT8→INT16 Matrix Multiplication (with bias): Speedup per configuration.

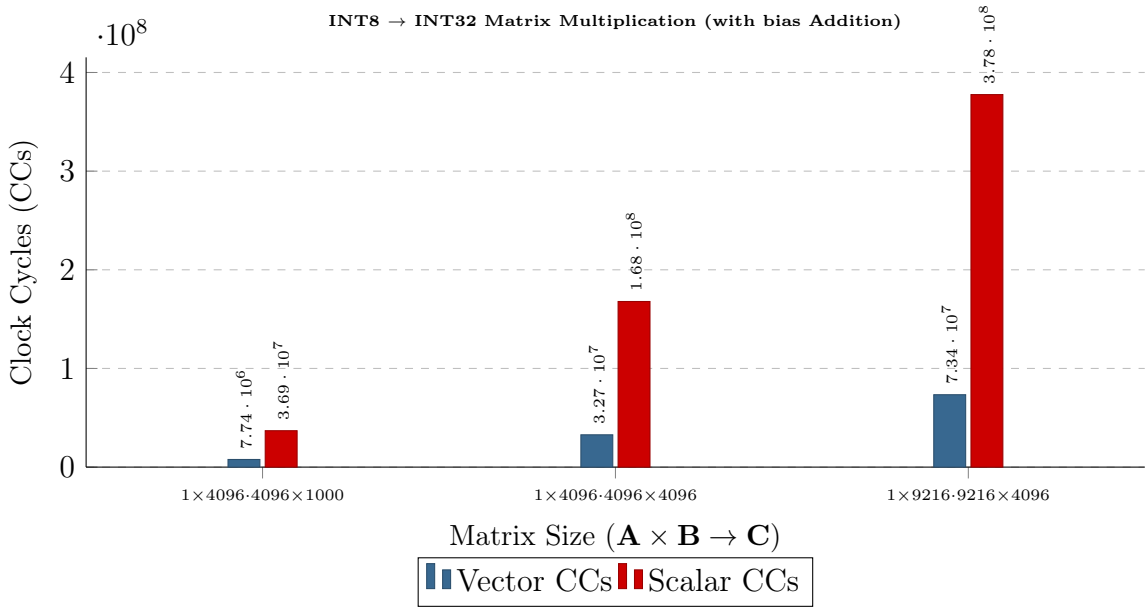

Figure 6.19: INT8→INT32 Matrix Multiplication (with bias): Vector vs. Scalar clock cycles.

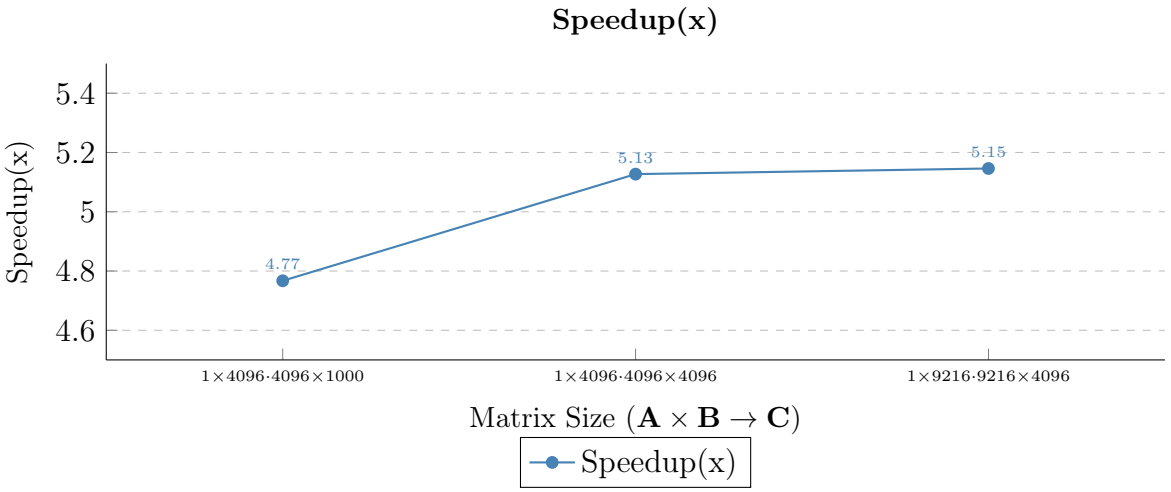

Figure 6.20: INT8→INT32 Matrix Multiplication (with bias): Speedup per configuration.

6.3.5 Performance Analysis

Across all ten configurations the reported speedup ranges from 4.688× to 9.115×. In Group 1 (int16, no bias) the speedups are tightly clustered between 6.051049× and 7.706078×. The highest value, 7.706078×, is achieved by the FFN-shaped configuration $A[8\times128]\cdot B[128\times512]$, whose wide output dimension ($N = 512$) allows the vector unit to process long contiguous segments of each **B**-row, maximizing lane utilization and minimizing the proportion of loop-control overhead. The lowest Group 1 value, 6.051049×, occurs for the square $A[64\times64]\cdot B[64\times16]$ case, whose narrow output dimension ($N = 16$) yields fewer full-width vector iterations before the loop tail, while the outer i- and k-loops (64 iterations each) contribute proportionally more overhead. The smallest configuration ($C[32\times8]$, 6.7728108×) is limited by its short inner dimension ($N = 8$) and its low total operation count, which provides limited opportunity for overhead amortization.

Group 2 isolates the effect of the shared dimension K while holding the output shape fixed at $C[32 \times 10]$. The speedup rises with K for the power-of-two values $32 \to 64 \to 128$ (4.688× → 6.246× → 9.115×), but the $K = 100$ configuration breaks this trend, falling to 5.255×, below both $K = 64$ and $K = 128$. The anomaly is visible directly in the cycle counts: the $K = 100$ vector cost (55,902 cycles) exceeds the $K = 128$ vector cost (45,535 cycles) despite $K = 100$ being the smaller shared dimension. A purely numeric observation is that 32, 64, and 128 are powers of two whereas 100 is not; however, the benchmark source provides no information that would explain the cause, so the origin of this non-monotonic behavior cannot be determined from the available data.

Group 3 contains three large matrix–vector products ($M = 1$) at int32 output precision. The speedup increases slightly with problem size and then saturates: 4.766997×, 5.126672×, and 5.145841×. The two largest cases differ by less than 0.4 percent in speedup (5.126672× vs. 5.145841×) even though the largest case (73,376,856 vector cycles) is roughly an order of magnitude larger than the smallest, indicating that the acceleration plateaus near approximately 5.1× in this regime.

The effect of output precision cannot be isolated from these data because no int16/int32 pair is provided at identical dimensions. Descriptively, the int32 group (4.766997×–5.145841×) sits below the int16 no-bias group (6.051049×–7.706078×), but because the dimensions and the bias setting also differ between the two groups, the gap cannot be attributed to output precision alone. Likewise, the effect of bias addition cannot be isolated, since the no-bias group (Group 1) and the with-bias group (Group 2) use different output shapes (Group 2 fixes $N = 10$) and no matched with/without-bias pair is provided. The scalar cycle counts grow approximately in proportion to $M \cdot K \cdot N$, consistent with a compute-bound scalar baseline.

6.3.6 Discussion

The benchmark results show that the vectorized matrix-multiplication kernel delivers consistent acceleration in the range of approximately 4.7× to 9.1× across all tested configurations. The implementation scales favorably with the output width N: the widest output ($N = 512$, the FFN case) yields the highest Group 1 speedup, consistent with the output-column parallelization strategy described in Section 6.3.2. The moderate speedups relative to purely element-wise operations reflect the inherent structure of matrix multiplication, the accumulation loop introduces a cross-iteration data dependency, and the column-wise access of **B** may incur stride penalties under a row-major layout. The clearest open question is the $K = 100$ anomaly in Group 2, which the available data does not explain and which would warrant additional measurements at non-power-of-two shared dimensions to characterize.

6.3.7 Key Observations

- Reported matrix-multiplication speedups span 4.688× to 9.115× across all configurations.
- int8→int16, no-bias speedups are tightly clustered (6.051049×–7.706078×); the FFN configuration ($N = 512$) is the highest at 7.706078×.
- With the output fixed at $C[32 \times 10]$, speedup increases with K for powers of two (4.688× → 6.246× → 9.115×) but is non-monotonic at $K = 100$ (5.255×); the cause is not present in the data.
- Large int32 matrix–vector products ($M = 1$) plateau near approximately 5.1× (4.766997×–5.145841×).
- Output-precision and bias effects cannot be isolated, because the source provides no matched (same-shape) comparisons.

6.4 Convolution

6.4.1 Overview

Convolution is the defining operation of convolutional neural networks (CNNs). It extracts spatially local features by sliding learned filters across multi-channel input feature maps, with each output element formed as a sum of products over the kernel window and the input channels. Its cost scales with the input spatial dimensions, the kernel size, and the input and output channel counts, which makes it one of the most expensive operations in a vision inference pipeline. This section reports vector-versus-scalar cycle counts for 2D convolution with 3×3, 5×5, and 7×7 kernels at two output precisions (int16 and int32), and for a depthwise 3×3 convolution.

6.4.2 Software Implementation

The scalar version computes each output element as a running sum of products over the kernel window and the input channels. The vector version parallelizes along the output width: it issues vector loads along each input row of the kernel patch and accumulates the weighted contributions across lanes using `vwmacc.vv`. The input datatype is int8; the accumulated output is stored as either int16 or int32 depending on the configuration. The int8→int16 path uses `vwmul.vv`/`vwmacc.vv` at SEW=8 widening to 16 bits, and the int8→int32 path widens in two stages, since the widening instructions only support SEW=8 and 16.

6.4.3 Benchmark Setup

The benchmark source records total clock cycles for the full convolution, for the scalar and vector builds of each configuration. Input dimensions, channel counts, stride, and padding are not recorded in the source and are therefore not stated here. The reported metric is total clock cycles, and the derived speedup is the scalar-to-vector cycle ratio.

6.4.4 Benchmark Results

3×3 convolution:
5×5 convolution:
7×7 convolution:
Depthwise 3×3 convolution:

Table 6.12: Conv2D 3×3 benchmark results.

Output type	Scalar Cycles	Vector Cycles	Speedup
int16	6,502,693	1,048,493	6.20×
int32	6,340,373	2,165,772	2.93×

Table 6.13: Conv2D 5×5 benchmark results.

Output type	Scalar Cycles	Vector Cycles	Speedup
int16	16,073,982	2,942,943	5.46×
int32	15,909,628	5,660,488	2.81×

Table 6.14: Conv2D 7×7 benchmark results.

Output type	Scalar Cycles	Vector Cycles	Speedup
int16	3,782,301	761,424	4.97×

Table 6.15: Depthwise Conv2D 3×3 benchmark results.

Output type	Scalar Cycles	Vector Cycles	Speedup
int16	503,402	149,912	3.36×

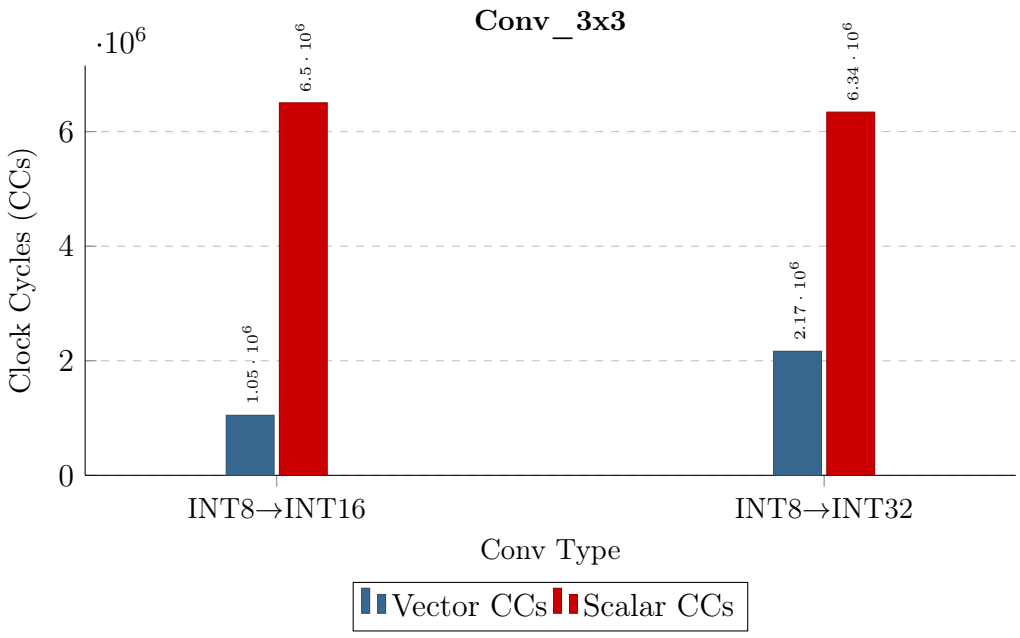

Figure 6.21: Conv2D 3×3: Vector vs. Scalar clock cycles.

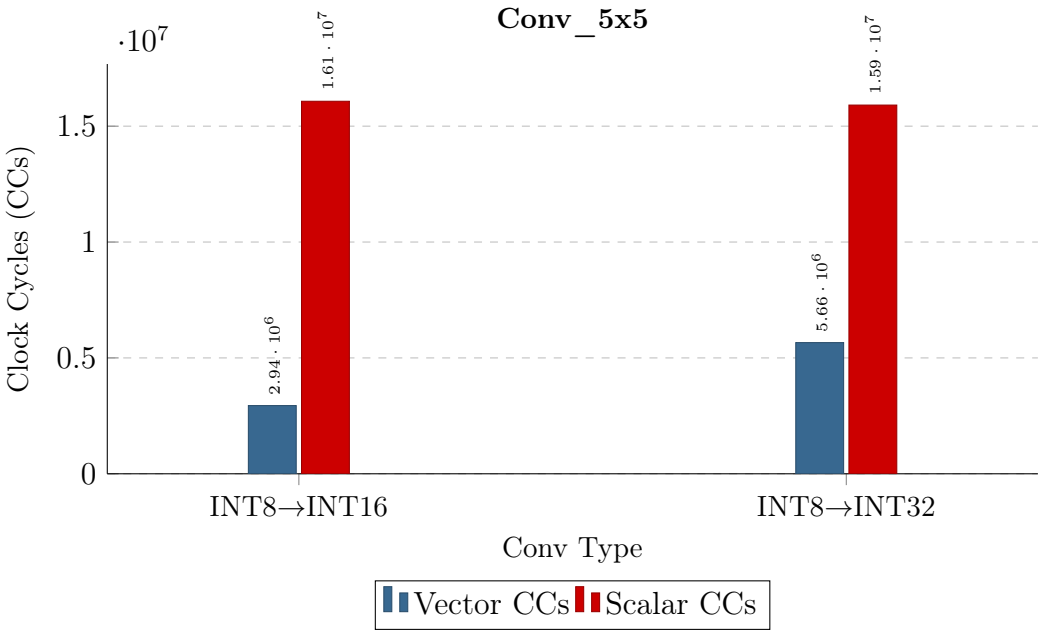

Figure 6.22: Conv2D 5×5: Vector vs. Scalar clock cycles.

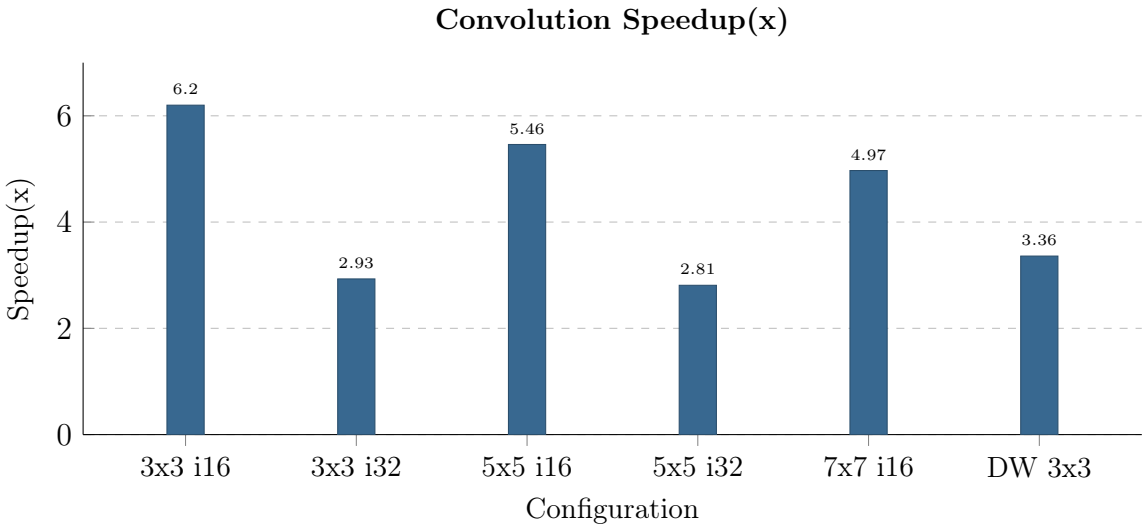

Figure 6.23: Convolution speedup across kernel sizes and output precisions.

6.4.5 Performance Analysis

Across the reported cases the speedup ranges from 2.81× (5×5, int32) to 6.20× (3×3, int16). Output precision has a clear and consistent effect for the kernels measured at both precisions. Moving from int16 to int32 lowers the speedup for both the 3×3 kernel (6.20× → 2.93×) and the 5×5 kernel (5.46× → 2.81×). The scalar cost is almost unchanged between precisions in both cases (6,502,693 → 6,340,373 for 3×3; 16,073,982 → 15,909,628 for 5×5), while the vector cost roughly doubles (1,048,493 → 2,165,772 for 3×3; 2,942,943 → 5,660,488 for 5×5). The int32 path widens in two stages because the widening instructions support only SEW=8 and 16, so producing a 32-bit accumulator costs roughly twice the vector work of the 16-bit path, which is what pulls the int32 speedup down.
The depthwise 3×3 convolution reaches 3.36×. A depthwise kernel does no accumulation across input channels, so it performs far less arithmetic per output element than a standard convolution, which lowers the ratio of vectorizable multiply-accumulate work to loop and addressing overhead.

6.4.6 Discussion

The vectorized convolution kernels accelerate the scalar baseline by between 2.81× and 6.20× across the tested cases. The int16 output paths consistently outperform the int32 paths on the same kernel, because the int32 accumulation requires a second widening stage under the SEW=8/16 constraint of the widening instructions. The depthwise variant sits below the standard 3×3 int16 result, consistent with its lower arithmetic intensity per output element.

6.4.7 Key Observations

- Convolution speedups span 2.81× to 6.20× across the tested cases.
- For a given kernel, int16 output outperforms int32 output (6.20× vs. 2.93× for 3×3; 5.46× vs. 2.81× for 5×5), because the int32 path needs a second widening stage under the SEW=8/16 limit of the widening instructions.
- The scalar cost barely changes with output precision, so the speedup drop at int32 comes almost entirely from the higher vector cost.
- Depthwise 3×3 reaches 3.36×, lower than standard 3×3 int16, consistent with its lower arithmetic intensity (no cross-channel accumulation).

6.5 Pooling

6.5.1 Overview

Pooling operations perform spatial downsampling of feature maps by aggregating values within a sliding window. Max pooling, which selects the maximum value within each pooling window, is the most widely employed variant, providing translation invariance and reducing the spatial resolution of feature maps between convolutional stages. In resource-constrained edge inference, efficient pooling is necessary to avoid making the downsampling step a bottleneck in the inference pipeline, particularly in networks with multiple pooling layers such as VGG and MobileNet variants.
The implemented operation is a 2×2 max pooling with stride 2 operating on int8 feature maps in channels-first $[C][H][W]$ format, targeting a vector configuration of VLEN=128, SEW=8,

and LMUL=2. Five real-application workloads from the MLPerf Tiny and standard CNN benchmark suites were evaluated to characterize the performance across realistic operating conditions.

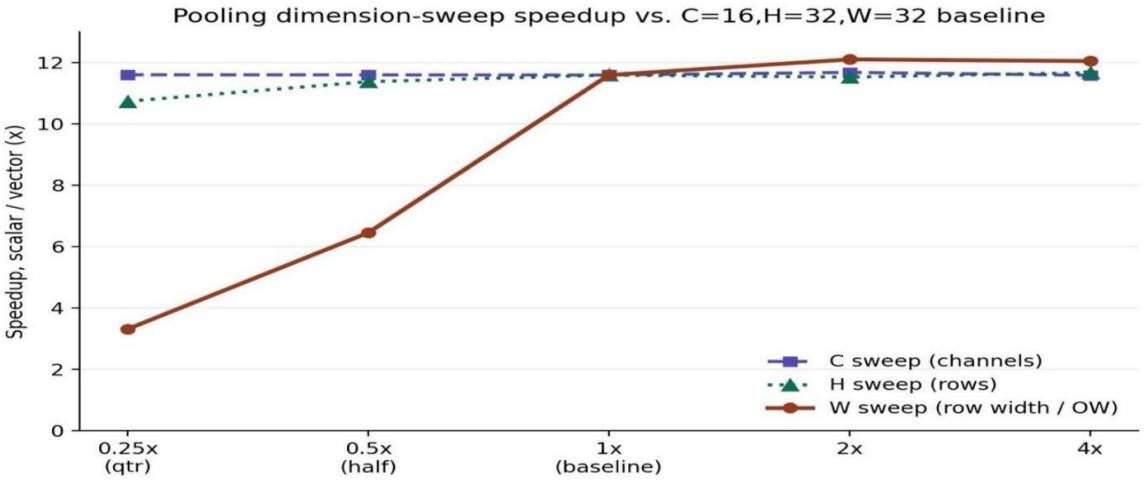

Figure 6.24: Pooling dimension-sweep speedup vs. $C = 16$, $H = 32$, $W = 32$ baseline. The W sweep (row width / OW) shows the strong dependence of speedup on output width alignment with VLMAX.

6.5.2 Software Implementation

The scalar implementation iterates over each output position (c, oh, ow), reads the four elements of the 2×2 pooling window from the input feature map, and writes the maximum to the output. The vector implementation parallelizes along the output width dimension (ow): for each output row, a vector of OW output elements is produced by loading two rows of the pooling window simultaneously (using two vector loads of width OW $\times$ stride), applying a vector element-wise maximum (`vmax.vv`), and storing the result. When OW $\leq$ VLMAX ($= 16$ for SEW=8, LMUL=2), the entire output row fits in a single vector register, eliminating the need for sub-row strip mining. When OW $>$ VLMAX, multiple vector passes are required per output row.

6.5.3 Benchmark Setup

Five application-representative benchmark configurations were evaluated, drawn from the CIFAR-10, MLPerf Tiny Visual Wake Words (VWW), MLPerf Tiny Keyword Spotting (KWS), Tiny Object Detection (FOMO), and Human Activity Recognition (HAR) workloads. Each configuration specifies the number of channels (C), input height (H), input width (W), output height (OH), and output width (OW). The OW value relative to VLMAX determines whether one or multiple vector passes per row are required.

6.5.4 Benchmark Results

Table 6.16: 2×2 max pooling (stride 2) benchmark results, CV32E40P + VXT (VLEN=128, SEW=8, LMUL=2, int8). Average speedup: 8.27×.

Application	$C \times H \times W$ **(Input)**	**OW**	**Vector**	**Scalar**	**Speedup**
CIFAR-10 VGG-small (Pool1)	$32 \times 32 \times 32$	16 (= VLMAX)	22,763	265,722	11.67×
VWW MobileNetV1 96×96 (Pool1)	$8 \times 48 \times 48$	24 (> VLMAX, ×2)	15,615	148,931	9.54×
KWS DS-CNN MFCC (Pool1)	$64 \times 24 \times 20$	10 (partial pass)	34,183	250,904	7.34×
FOMO Tiny Object Det. (Pool1)	$16 \times 26 \times 26$	13 (odd-width tail)	9,255	87,837	9.50×
HAR IMU CNN Wearable (Pool1)	$32 \times 16 \times 8$	4 (very narrow)	10,475	34,533	3.30×

6.5.5 Performance Analysis

The pooling benchmarks reveal a strong dependence of speedup on the output width (OW) relative to VLMAX. The highest speedup of 11.67× is achieved by the CIFAR-10 configuration ($C = 32$, $H = W = 32$, OW = 16), where OW equals exactly VLMAX. In this case, each output row is processed in precisely one full-width vector pass, with no tail-handling overhead and no strip-mining branches. The vector unit operates at 100% lane utilization for all output rows, yielding near-optimal efficiency.

The VWW configuration (OW = 24 > VLMAX) achieves 9.54×, requiring two vector passes per output row. The second pass covers the remaining $24 - 16 = 8$ elements with a reduced vector length, introducing tail-handling overhead proportional to the number of output rows (OH = 24). This explains the slight reduction in speedup relative to CIFAR-10.

The FOMO configuration (OW = 13) and KWS configuration (OW = 10) represent non-power-of-2 output widths below VLMAX, where each output row requires a single partial vector pass with a non-aligned tail. The KWS configuration ($C = 64$, OW = 10) achieves only 7.34×, the second-lowest speedup, because the large channel count (64) multiplies the tail-handling overhead across all $C = 64$ channels and OH = 12 rows, resulting in a high proportion of partial-length vector operations.

The HAR configuration (OW = 4) achieves the lowest speedup of 3.30×. With OW = 4, each output row requires a vector operation covering only 4 of VLMAX=16 lanes, a 75% waste of vector capacity. The majority of each vector operation's execution time is attributable to fixed instruction overhead (`setvl`, load, `vmax`, store) rather than productive computation, explaining the poor speedup. This represents a fundamental limitation of the implementation for very narrow feature maps.

The average speedup across all five applications is 8.27×, reflecting a practically significant acceleration for the pooling operation across a diverse set of real-world workloads.

6.5.6 Discussion

The pooling speedup is strongly correlated with the alignment between the output width and the vector width. The comparison demonstrates that workloads where OW is a multiple of VLMAX (or equals VLMAX) achieve the highest speedup, while narrow feature maps

(OW $\ll$ VLMAX) suffer from low lane utilization. This observation has practical implications for network architecture design targeting this platform: layers whose output spatial dimensions align with VLMAX will execute pooling significantly faster. For the CV32E40P + VXT configuration (VLMAX = 16 for SEW=8, LMUL=2 used for pooling output), output widths that are multiples of 16 are optimal.

6.5.7 Key Observations

- CIFAR-10 (OW = 16 = VLMAX) achieves the highest speedup of 11.67×, confirming that OW-alignment with VLMAX is the primary performance driver.
- HAR (OW = 4 $\ll$ VLMAX) achieves the lowest speedup of 3.30×, demonstrating the performance penalty for very narrow feature maps.
- The average speedup across all five applications is 8.27×, representing a practically significant reduction in inference latency.
- Non-power-of-2 output widths (OW=10, OW=13) introduce tail-handling overhead that reduces speedup relative to aligned configurations.
- For configurations with OW > VLMAX, the number of vector passes per row determines the speedup penalty relative to the OW = VLMAX ideal.

6.6 Software Benchmark Summary

This section synthesizes the benchmark results across all implemented software kernels and presents a consolidated analysis of speedup trends, vectorization effectiveness, and computational characteristics.

6.6.1 Overall Speedup Comparison

6.6.2 Functions Benefiting Most from Vectorization

The functions that benefit most from vectorization are those with the highest ratio of data parallelism to control-flow complexity, and the most favorable memory access patterns for sequential vector loads and stores. ReLU is the clearest example: it is a single-instruction operation (`vmax.vx`) with no data-dependent branches, operating on int8 data (the widest VLMAX), with perfectly sequential access patterns. It achieves the highest observed speedup of 31.25×.
Requantization, particularly int16 → int8, also achieves very high speedups (up to 21.75×) because the narrowing saturating right shift instruction subsumes multiple scalar operations (shift, clamp, store) into a single vector instruction, and the output data type is narrower, increasing the number of elements processed per store instruction.
Pooling achieves high speedup (up to 11.67×) when the output width aligns with VLMAX, reflecting the efficiency of vector element-wise maximum over contiguous memory segments.

6.6.3 Computationally Intensive Operations

Matrix multiplication is the most computationally intensive operation in terms of absolute cycle counts, with the FFN configuration ($A[8 \times 128] \times B[128 \times 512]$) requiring 5,271,628 scalar cycles. Softmax for large classification vocabularies (N = 32,000) requires 2,437,809 scalar

Table 6.17: Summary of peak and minimum speedups for all benchmarked software functions.

Function / Variant	**Peak Speedup**	**Min Speedup**	**Primary Bottleneck**
ReLU (int8)	31.25×	20.22×	Strip-mining overhead at small N
int16→int8 Requantization	21.75×	19.99×	Per-call overhead at $N = 128$
int32→int8 Requantization	14.45×	13.02×	Two-stage narrowing
int32→int16 Requantization	10.65×	9.70×	Wider input type (SEW=32)
Pooling (CIFAR-10, OW=16)	11.67×	3.30×	OW alignment with VLMAX
Softmax (Profile A, $N = 1024$)	8.50×	2.54×	Very small N, reduction overhead
Sigmoid (Profile B)	6.62×	5.96×	PLA complexity, 5 segments
Matrix Multiplication (FFN)	7.71×	6.05×	Output column width
GELU (Profile A)	5.13×	4.95×	PLA complexity, 5 segments
Conv2D 3×3 (int16)	6.20×	6.20×	Widening MAC across output width
Conv2D 3×3 (int32)	2.93×	2.93×	Two-stage widening to 32-bit
Conv2D 5×5 (int16)	5.46×	5.46×	Kernel footprint, widening MAC
Conv2D 5×5 (int32)	2.81×	2.81×	Two-stage widening to 32-bit
Conv2D 7×7 (int16)	4.97×	4.97×	Larger kernel footprint
Depthwise Conv 3×3	3.36×	3.36×	No cross-channel accumulation

cycles. Conv2D with 5×5 kernels at int16 output requires 16,073,982 scalar cycles, the highest absolute cycle count of all benchmarked configurations, reflecting the $O(H \cdot W \cdot K^2 \cdot \mathrm{CH_{in}} \cdot \mathrm{CH_{out}})$ complexity scaling of 2D convolution with large kernels.
Despite their high absolute cost, these operations achieve meaningful vector speedups (2.81–6.20× for convolution, 7.21–8.50× for softmax, 6.05–7.71× for matrix multiplication), showing that the VXT vector unit substantially reduces their latency even for the heaviest workloads.

6.6.4 Effectiveness of Software Optimizations

The principal optimizations employed across the benchmarked implementations are: (1) strip-mining loops with tail handling for arbitrary-length inputs; (2) branchless mask-based evaluation for piecewise linear functions (GELU, Sigmoid, Softmax); (3) fixed-point Q3.12 arithmetic with `vmulh` for multiplication without overflow; (4) narrowing saturating right shift (`vnsra`) for requantization; (5) broadcast scalar operands for convolution accumulation and activation thresholds; and (6) alignment of data structures to VLMAX boundaries where applicable.
The most impactful single optimization across all functions is the elimination of data-dependent branches in the vector path. All PLA-based functions (GELU, Sigmoid, Softmax) use mask-based segment selection via `vmerge`, replacing the scalar if-else chain with branchless merge instructions. This eliminates branch misprediction overhead entirely in the vector path, contributing to the observed speedup in proportion to the fraction of execution time attributable to branch resolution in the scalar implementation.

6.6.5 Overall Conclusions

The benchmark results show that the CV32E40P + VXT vector platform delivers substantial acceleration across the full spectrum of neural network inference kernels, from simple element-wise operations to complex piecewise nonlinear functions and spatially structured convolutions. The observed speedups span a range from approximately 2.81× (depthwise-style low-intensity convolution) to 31.25× (ReLU at 65,536 elements), with most functions achieving speedups in the range of 5–12× under representative workloads.
The results confirm several general principles of vectorized inference optimization. First, element-wise operations on narrow data types (int8) benefit most from vectorization, approaching the theoretical VLMAX-fold speedup. Second, operations with PLA-based nonlinear evaluation achieve moderate but consistent speedups (5–8×) across all input sizes, with the speedup governed by the ratio of vector instruction count to VLMAX rather than by data type alone. Third, spatially structured operations (convolution, pooling) exhibit strong sensitivity to the alignment between spatial dimensions and vector width, with misalignment significantly degrading efficiency. Fourth, requantization, a frequently overlooked operation, achieves among the highest speedups in the benchmark suite, highlighting it as a primary optimization target in quantized inference pipelines.
Overall, the VXT vector unit enables a practical inference acceleration factor in the range of 6–10× across a balanced neural network workload, with peak speedups exceeding 30× for the most vectorization-friendly kernels.

6.6.6 Key Observations, Summary

- ReLU achieves the highest speedup (31.25×), approaching the VLMAX=32 theoretical limit for int8 data under LMUL=2.
- int16→int8 requantization achieves the second-highest speedup (21.75×), driven by the narrowing saturating right shift instruction eliminating explicit saturation logic.

- Softmax achieves the highest speedup among PLA-based nonlinear functions (8.50×), benefiting from vectorized reduction operations.
- Conv2D 3×3 at int16 achieves the highest convolution speedup (6.20×), with int16 output consistently outperforming int32 across kernel sizes.
- Depthwise 3×3 convolution achieves 3.36×, lower than standard 3×3 because it does no cross-channel accumulation and so has lower arithmetic intensity per output element.
- Pooling speedup ranges from 3.30× (HAR, OW=4) to 11.67× (CIFAR-10, OW=16=VLMAX), confirming OW-VLMAX alignment as the dominant pooling performance factor.
- The average pooling speedup of 8.27× across five real-application workloads demonstrates practical, application-representative acceleration.
- Branchless mask-based evaluation (`vmerge`) is the key enabler for PLA-based functions, eliminating scalar branch misprediction overhead entirely in the vector path.

Chapter 7

Model Benchmarks

Contents

7.1 Introduction 146
7.2 Benchmark Metrics 146
7.3 Visual Wake Words MobileNetV1 Benchmark 146
7.3.1 Model Overview and Usage 146
7.3.2 Reason for Selection 146
7.3.3 Implemented Structure 147
7.3.4 Benchmark Results 147
7.4 GPT-2 Single Decoder Block Benchmark 148
7.4.1 Model Overview and Usage 148
7.4.2 Reason for Selection 148
7.4.3 Implemented Structure 149
7.4.4 Benchmark Results 149
7.5 Overall Benchmark Comparison 150
7.6 Chapter Summary 151

7.1 Introduction

After validating the proposed vector extension at the instruction level and evaluating the acceleration of the individual computational kernels, this chapter evaluates the complete model-level benchmarks. The purpose of this chapter is to show the effect of the implemented RISC-V vector support on full artificial intelligence workloads, rather than isolated kernels only.
The benchmarked models are Visual Wake Words MobileNetV1 and one GPT-2 style decoder block. These models were selected to cover two different workload classes. MobileNetV1 for Visual Wake Words represents a lightweight embedded-vision workload based on depthwise separable convolution. The GPT-2 decoder block represents a transformer-based workload dominated by matrix multiplication, attention, normalization, and feed-forward operations.
For each model, two implementations are compared: a scalar implementation and a vectorized implementation using the implemented RISC-V vector kernels. The benchmark methodology, compilation flow, memory-loading procedure, RTL simulation environment, and scalar-versus-vector output comparison were described in previous chapters. Therefore, this chapter focuses on the model structures, benchmark motivation, and final cycle-count results.

7.2 Benchmark Metrics

The main metric used in this chapter is the total number of clock cycles required to execute the full model. The scalar implementation is used as the baseline, while the vector implementation represents the accelerated version. The acceleration is calculated as:

$$\text{Acceleration} = \frac{\text{Scalar cycle count}}{\text{Vector cycle count}} \tag{7.1}$$

A higher acceleration value means that the vector implementation completes the model execution in fewer cycles compared to the scalar baseline.

7.3 Visual Wake Words MobileNetV1 Benchmark

7.3.1 Model Overview and Usage

The Visual Wake Words model is a lightweight image-classification workload used for person-detection applications. The model classifies an input image into two classes: person and non-person. In this work, the benchmark uses a MobileNetV1-based Visual Wake Words model with width multiplier $\alpha = 0.25$ and an input size of $96 \times 96 \times 3$.
MobileNetV1 is based on depthwise separable convolution. Instead of using a standard convolution for every layer, it separates the convolution into two operations: a depthwise 3×3 convolution applied independently to each channel, followed by a pointwise 1×1 convolution used to combine information across channels. This structure significantly reduces computation compared to standard convolution and is commonly used in embedded and low-power vision applications.

7.3.2 Reason for Selection

The VWW MobileNetV1 model was selected because it represents a realistic tiny-ML and embedded-vision workload. MobileNetV1 is designed to be computationally efficient, which makes it suitable for evaluating whether the implemented vector extension is useful for compact networks that are closer to edge-AI deployment.

This benchmark is also important because it contains many depthwise and pointwise convolution layers. These operations have different memory-access and arithmetic patterns compared to standard convolution, which makes the model useful for evaluating the generality of the vectorized kernel library.

7.3.3 Implemented Structure

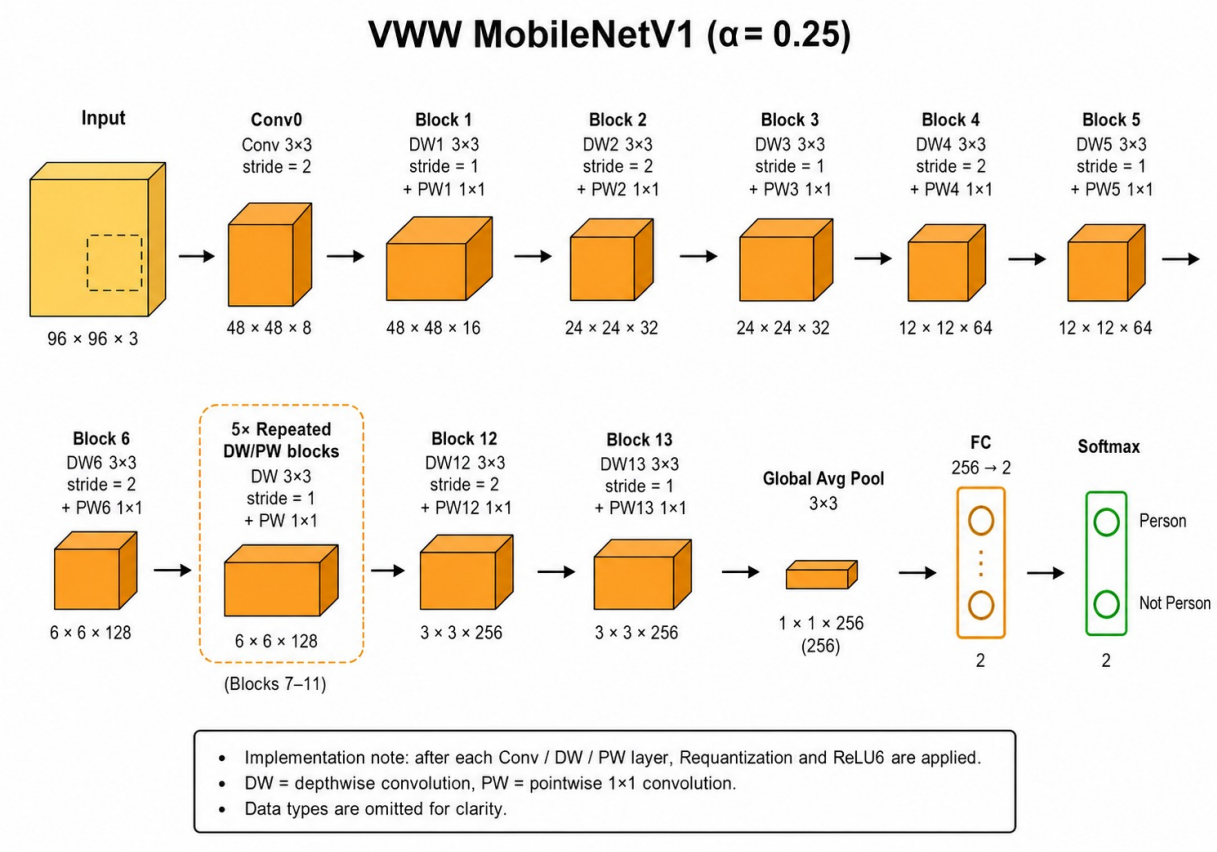

Figure 7.1: Visual Wake Words MobileNetV1 architecture used in the implemented benchmark flow.

7.3.4 Benchmark Results

Table 7.1: VWW MobileNetV1 benchmark results.

Metric	Value
Scalar cycle count	293,801,969
Vector cycle count	62,103,469
Acceleration	4.731×

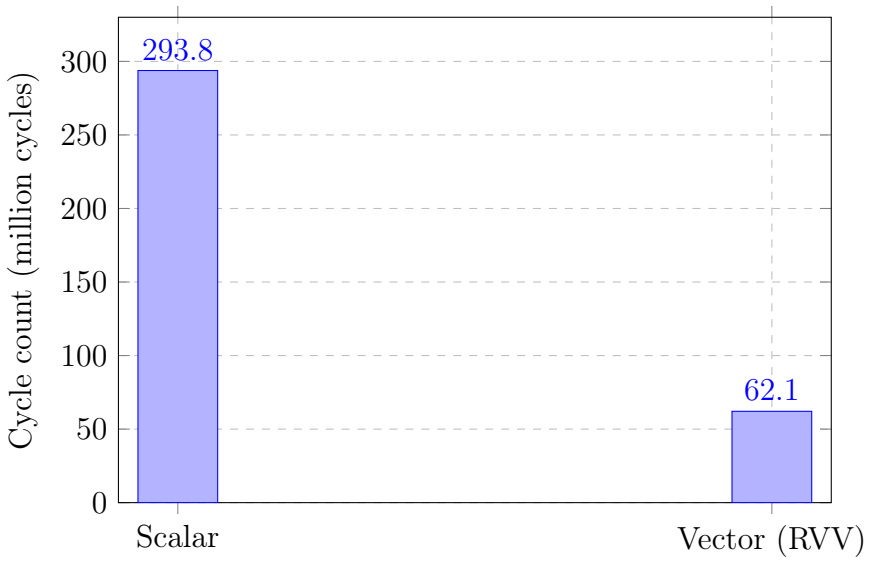

Figure 7.2: VWW MobileNetV1 benchmark cycle counts.

The acceleration of VWW MobileNetV1 is:

$$\text{Acceleration}_{\text{MobileNet}} = \frac{293{,}801{,}969}{62{,}103{,}469} = 4.731\times \tag{7.2}$$

The MobileNet benchmark evaluates the proposed vector implementation on a lightweight model where the computation is distributed across many small depthwise and pointwise convolution stages. The total acceleration depends not only on the arithmetic acceleration of the convolution kernels, but also on the memory movement, requantization, ReLU6 operations, and the overhead of many small layers.

7.4 GPT-2 Single Decoder Block Benchmark

7.4.1 Model Overview and Usage

GPT-2 is a transformer-based language model used for natural-language-processing tasks such as text generation and sequence modeling. Unlike convolutional neural networks, GPT-2 is based on decoder-only transformer blocks. These blocks mainly depend on layer normalization, matrix multiplication, masked self-attention, residual connections, nonlinear activation, and feed-forward neural-network layers.
In this benchmark, one GPT-2 style decoder block is evaluated. The benchmark starts from preloaded input hidden states and runs a single decoder layer. Token embedding, positional embedding, the repeated stack of decoder blocks, and the final language-model head are outside the scope of this benchmark.

7.4.2 Reason for Selection

The GPT-2 decoder block was selected to evaluate the proposed vector implementation on transformer workloads. This is important because transformers have become dominant in modern AI applications, while their computation pattern is different from CNNs.
The GPT-2 decoder block stresses matrix multiplication in the QKV projection, attention score calculation, attention output computation, attention output projection, and feed-forward network. By including this benchmark, the evaluation is not limited to computer-vision CNNs.

It also demonstrates whether the vector extension can accelerate attention-based and dense linear-algebra workloads.

7.4.3 Implemented Structure

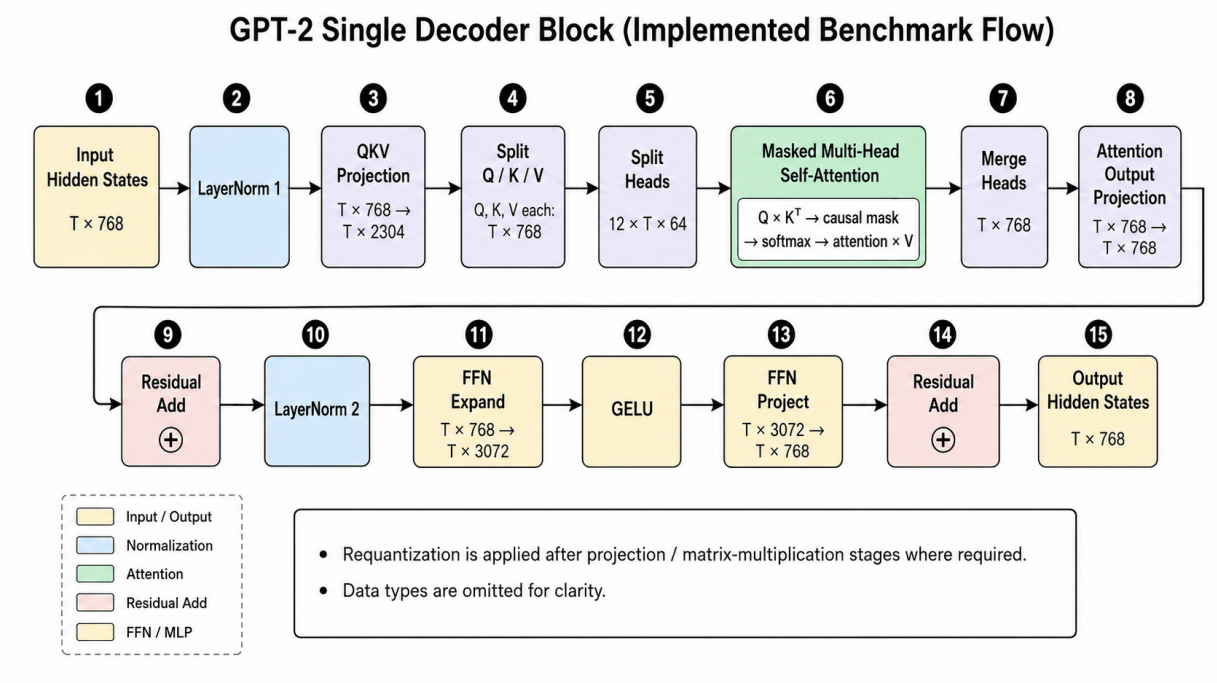

Figure 7.3: GPT-2 single decoder block architecture used in the implemented benchmark flow.

7.4.4 Benchmark Results

Table 7.2: GPT-2 single decoder block benchmark results.

Metric	Value
Scalar cycle count	1,147,471,952
Vector cycle count	163,923,573
Acceleration	7.000×

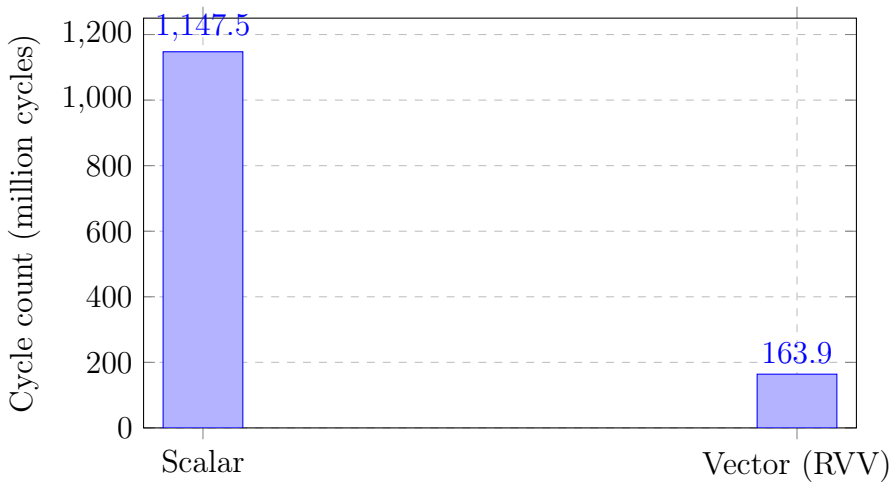

Figure 7.4: GPT-2 decoder block benchmark cycle counts.

The acceleration of the GPT-2 decoder block is:

$$\text{Acceleration}_{\text{GPT2}} = \frac{1{,}147{,}471{,}952}{163{,}923{,}573} = 7.000\times \tag{7.3}$$

The GPT-2 decoder block benefits from vector execution because the QKV projection, attention score computation, attention output computation, output projection, and feed-forward network are dominated by matrix multiplication. However, the final model-level acceleration also depends on the cost of split/merge operations, transposition, causal masking, softmax, requantization, GELU, residual addition, and memory movement.

7.5 Overall Benchmark Comparison

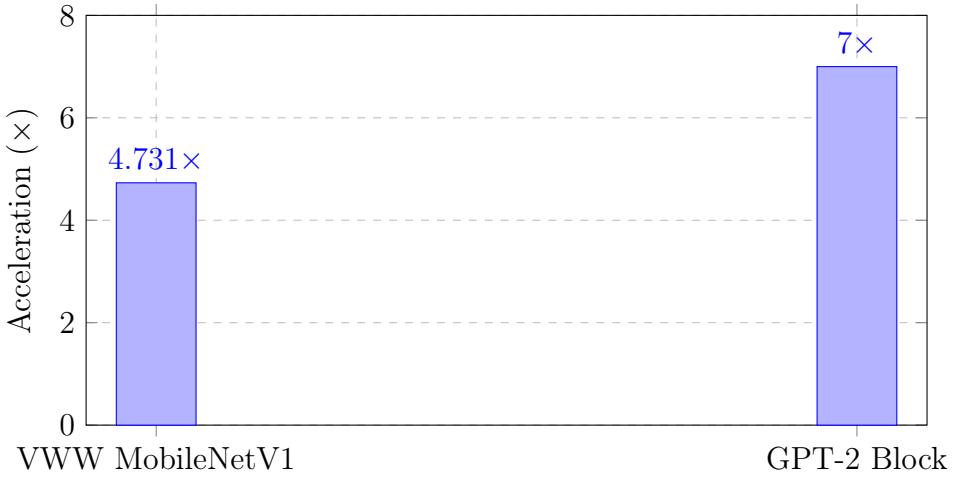

Figure 7.5: Acceleration comparison for the model benchmarks.

The overall comparison shows how the proposed vector implementation behaves across two different AI workload classes. VWW MobileNetV1 evaluates an embedded vision model based on depthwise separable convolution. The GPT-2 decoder block evaluates a transformer workload dominated by attention and feed-forward matrix multiplication.
The achieved acceleration for each model depends on how much of the total execution time is spent in vectorizable arithmetic kernels. Layers dominated by multiply-accumulate operations

usually benefit more from vector execution. On the other hand, operations such as padding, pooling, split/merge, softmax, residual addition, and memory movement may reduce the overall model-level acceleration compared to the acceleration observed in isolated kernels.

7.6 Chapter Summary

This chapter presented the model-level benchmark evaluation of the proposed vector implementation. Two models were selected: VWW MobileNetV1 and one GPT-2 style decoder block. These benchmarks cover lightweight depthwise-separable CNN computation and transformer-based attention computation.

The VWW MobileNetV1 benchmark achieved an acceleration of 4.731×, reducing the cycle count from 293,801,969 scalar cycles to 62,103,469 vector cycles. The GPT-2 decoder block achieved an acceleration of approximately 7.000×, reducing the cycle count from 1,147,471,952 scalar cycles to 163,923,573 vector cycles. These results show that the implemented RISC-V vector support improves performance across both embedded vision and transformer-style workloads.

Chapter 8

Conclusion, Limitations, and Future Work

Contents

8.1 Summary of the Work . . . 153
8.2 Key Contributions . . . 153
8.3 Key Results . . . 154
8.4 Limitations . . . 154
8.5 Future Work . . . 155
8.5.1 Architecture . . . 155
8.5.2 ISA . . . 155
8.5.3 Software . . . 156
8.5.4 Evaluation . . . 156
8.6 Conclusion . . . 156
8.7 Chapter Summary . . . 157

8.1 Summary of the Work

The core question this project started with was concrete: can a small, open-source RISC-V scalar processor accelerate edge AI inference without a GPU or a purpose-built neural-network chip, and how much faster does it actually run? The answer required building the full stack ourselves, from the RTL of the vector co-processor up through the kernel library and the model benchmarks.

We designed and implemented a custom RISC-V Vector extension (VXT) on the CV32E40P, an open-source 32-bit in-order core from the OpenHW Group. VXT implements a targeted subset of the RVV 1.0 specification, selected specifically for the operations that AI inference kernels actually need: widening multiply-accumulate, reductions, piecewise-linear function evaluation, and element-wise comparison and merge. The hardware runs at VLEN=128, supports SEW$\in \{8, 16, 32\}$ and LMUL$\in \{1, 2\}$, and is non-pipelined, meaning it processes one vector instruction to completion before accepting the next. That is a deliberate simplification that made the design tractable at graduation-project scale without compromising functional correctness.

From the RTL of the co-processor, we built a six-block microarchitecture (Decoder, VCSR, Controller, VREG, VALU, VLSU), verified each block in isolation with SystemVerilog testbenches, verified all supported instructions on the integrated accelerator system, and then validated the full kernel library against a QEMU functional reference using a three-way scalar-vs-vector-vs-golden comparison. The kernel library covers convolution (3×3, 5×5, 7×7, depthwise), matrix multiplication, pooling, requantization, and nonlinear activations including ReLU, sigmoid, GELU, and softmax. Model-level benchmarks on VWW MobileNetV1 and a GPT-2 decoder block closed the evaluation loop from individual operations up to complete inference tasks.

8.2 Key Contributions

1. **A verified RVV co-processor attached to CV32E40P.** We modified the CV32E40P decode stage to recognize the VXT opcode and stall the scalar pipeline, added an OBI bus multiplexer to arbitrate memory between the scalar LSU and the vector VLSU, and integrated the full VXT co-processor without changes to the host core beyond those two points. The RTL is open, modifiable, and verified at the instruction level.

2. **A self-consistent ISA reference.** Chapter 3 documents every supported instruction, its encoding, legal configurations, and mandatory programming rules. It also catalogs twelve deliberate divergences from the RVV 1.0 specification and one hardware erratum (D-12: `vwmacc.vx` scalar operand sign-extension, with its `vmv.v.x` + `vwmacc.vv` workaround). This reference exists so that anyone continuing this work has an accurate description of what the hardware actually does, not what the full spec requires.

3. **A quantized integer kernel library.** All kernels operate entirely in integer arithmetic, consistent with the int8 inference datapath used in practice. The library covers the computational primitives needed by both CNN and transformer inference, and each kernel was validated against a golden reference before its speedup was reported.

4. **An automated verification flow.** We built a fixed-address linker convention, a QuestaSim dump script (`verify.do`), and a Python comparison tool (`compare_outputs.py`) that together make kernel verification repeatable from the command line. The three-way comparison (scalar vs. vector vs. QEMU golden) catches hardware bugs, software bugs, and mismatches between the two independently.

5. **Model-level benchmarks on two workload classes.** The MobileNetV1 VWW and GPT-2 decoder benchmarks cover both CNN inference (dominated by depthwise and pointwise convolution) and transformer inference (dominated by matrix multiplication and attention). The cycle counts come from RTL simulation, not from QEMU, which has no timing model for the VXT pipeline.

8.3 Key Results

Table 8.1 summarizes the kernel-level speedups. Table 8.2 gives the model-level results.

Table 8.1: Summary of kernel-level speedups. All cycle counts from QuestaSim RTL simulation.

Kernel	**Best speedup**	**Note**
ReLU	up to 31.25×	Highest across all tested configs
Max Pooling (avg.)	8.27×	Avg. over 5 application configs
Softmax	2.54–8.50×	Varies with input length N
Sigmoid	5.96–6.62×	5-segment PLA, two profiles
GELU	4.95–5.13×	5-segment PLA, two profiles
Matrix Multiplication	6.05–7.71×	Varies with output column width
Conv2D 3×3 (int16)	6.20×	
Conv2D 3×3 (int32)	2.93×	Two-stage widening
Conv2D 5×5 (int16)	5.46×	
Conv2D 5×5 (int32)	2.81×	Two-stage widening
Conv2D 7×7 (int16)	4.97×	
Depthwise Conv 3×3	3.36×	Low arithmetic intensity

Table 8.2: Model-level benchmark results.

Model	**Scalar cycles**	**Vector cycles**	**Speedup**
VWW MobileNetV1	293,801,969	62,103,469	4.731×
GPT-2 decoder (1 block)	1,147,471,952	163,923,573	7.000×

A few things worth noting in these numbers. The int16 convolution results consistently beat the int32 results on the same kernel, because the VXT widening instructions only support SEW=8 and 16 as source widths, so producing a 32-bit accumulator requires two widening stages. The pooling speedup is highly sensitive to whether the output width is a multiple of VLMAX=16; the CIFAR-10 configuration (OW = 16 = VLMAX) achieves 11.67×, while the narrowest configuration (OW = 4) reaches only 3.30×. At the model level, the GPT-2 decoder outperforms MobileNetV1 partly because its workload is dominated by large matrix multiplications, where the vector unit's throughput advantage is cleanest.

8.4 Limitations

We want to be specific about what this project does and does not show.

No pipeline. VXT serializes instructions: each vector instruction runs to completion before the next begins. The non-pipelined design is what made the control unit tractable to build and verify in the time available, but it means the hardware does not approach the peak throughput

a pipelined unit would deliver. LMUL=2 is sometimes slower than LMUL=1 on this implementation (documented in the pooling results) because the extra register-beat overhead costs more than the additional parallelism saves under the serial execution model.

Software must manage latency. Because the hardware has no interlocks, every kernel must insert a `VXT_FENCE` instruction between a vector instruction that writes a register and the next instruction that reads it. This is a real co-design burden: any bug in the fence placement produces a silent wrong result rather than an exception. The fence pattern adds instruction overhead that slightly reduces effective throughput.

No area or power characterization. This project measured clock cycles. We have not synthesized VXT to an FPGA or ASIC target and have no area or power numbers to report. Whether the silicon cost of VXT is acceptable for a real edge device is a question this project cannot answer.

QEMU does not model pipeline timing. QEMU simulates the functional behavior of the RISC-V ISA correctly, which is why we use it as the golden reference for output verification. It has no timing model for the VXT multi-cycle latency. Any speedup numbers derived from QEMU execution would be meaningless; all the numbers in Chapters 6 and 7 come from QuestaSim RTL simulation.

SEW=32 widening requires two stages. The widening instructions (`vwmul`, `vwmacc`) only support SEW=8 and 16 as source widths. Accumulating int8 products into a 32-bit accumulator requires widening first to int16 and then to int32, which roughly halves the speedup compared to int16 output. This is a known constraint of the current ISA subset, not a hardware bug.

8.5 Future Work

8.5.1 Architecture

Pipeline the vector unit. This is the single change with the highest impact. A pipelined execution unit would issue a new vector instruction every cycle (or every few cycles), rather than stalling until the current instruction finishes. It would also make LMUL=2 and higher LMUL values genuinely useful, because the per-beat overhead would be amortized across concurrent execution rather than paid serially. Almost everything else on this list becomes more valuable once the unit is pipelined.

Hardware interlocks. With a pipelined unit, adding hazard detection so the hardware can stall itself on true data dependencies would eliminate the manual `VXT_FENCE` requirement. That drops a significant co-design burden from the kernel developer and removes a class of subtle wrong-result bugs.

Higher and fractional LMUL. LMUL$\in \{4, 8\}$ would let a single instruction process 64 or 128 int8 elements at once. Fractional LMUL (1/2, 1/4) would support workloads with very short vectors without wasting register bandwidth. Neither makes much sense until the unit is pipelined.

Larger VLEN. Moving from VLEN=128 to VLEN=256 or 512 doubles or quadruples the elements processed per beat. Combined with pipelining, this is the straightforward path to higher throughput at the cost of larger register file area.

8.5.2 ISA

Float and bf16 support. VXT is integer-only. Real transformer inference increasingly uses 16-bit floating point (fp16 or bf16) for intermediate computations. Adding a float ALU path to the existing VALU datapath would open these workloads.

Gather and scatter. Indexed loads (`vluxei`, `vloxei`) and stores let a vector instruction fetch non-contiguous elements from memory using a vector of byte offsets. They are needed for attention weight gathering in a full transformer pass and for some sparse-access patterns in embedding tables.
Direct SEW=32 widening. Extending the widening instructions to support int8→int32 in a single pass would remove the two-stage workaround and lift the int32 convolution speedup closer to the int16 baseline.

8.5.3 Software

Compiler autovectorization. Every kernel in this project is hand-written assembly or compiler intrinsics. Writing and maintaining the kernel library by hand is expensive and error-prone. GCC support for the VXT ISA subset would let the compiler generate vector code from ordinary C, which is the standard deployment model for production edge AI stacks.
A higher-level kernel API. The current kernels expose raw RVV intrinsics directly. A thin C API with functions like `vec_conv2d_int8()` and `vec_matmul_int8()` would let higher-level code (a model inference framework, for instance) target VXT without knowing the instruction set.
Broader model and layer coverage. The benchmark suite covers two models. Adding end-to-end attention (not just the matmul component), normalization layers (layer norm, batch norm), and embedding lookup would close the remaining gaps between the kernel library and a complete inference stack.

8.5.4 Evaluation

FPGA synthesis and characterization. The logical next step after functional verification is synthesis: place-and-route VXT on a representative FPGA target, measure the achievable clock frequency, and compare the resource cost (LUTs, flip-flops, BRAM) against the performance gain. This gives the number that this project cannot: whether VXT is a good area-performance trade-off for real deployment.
Power measurement. Dynamic power consumption under realistic AI workloads would complete the picture. A cycle-accurate simulation with power annotation would work before silicon; a real FPGA implementation with a power meter would be better.
Broader model coverage. AlexNet (Chapter 7 benchmarked the kernel components but not the full model), ResNet variants, and a full GPT-2 pass (not just one decoder block) are natural extensions of the existing benchmark flow.

8.6 Conclusion

We set out to find out whether a small, open, extensible RISC-V core can meaningfully accelerate edge AI without specialized hardware. The answer is yes, within the scope of what we built and measured.
A targeted subset of the RISC-V Vector extension, implemented as a co-processor on the CV32E40P, reduced execution time for VWW MobileNetV1 from 293 million scalar cycles to 62 million vector cycles, a factor of 4.73×. For a GPT-2 decoder block the reduction was from 1.15 billion to 164 million cycles, a factor of 7.00×. At the kernel level the picture is more varied: ReLU reaches 31× because its arithmetic is almost pure lane throughput, while narrow depthwise convolutions stay around 3× because the vector unit spends a significant fraction of its time on addressing overhead relative to the actual MAC work. These numbers are honest in the sense that they come from RTL simulation, they account for all the overhead of the

real kernel implementations, and the implementations were validated against an independent golden reference before the cycle counts were reported.
The hardware as built has real constraints. It is not pipelined. It has no interlocks. The scalar side of `vwmacc.vx` does not sign-extend correctly. Widening to int32 in a single pass is not supported. Area and power are unmeasured. We have documented all of these in the text so that whoever continues this work starts from an accurate picture of what the hardware does, not an idealized one.
What this project shows is that the combination of an open ISA, an open RTL baseline, and a constrained but well-chosen vector instruction subset is enough to deliver substantial inference acceleration at a level of effort that fits a graduation project. The remaining gap between this prototype and a production-ready accelerator is real, and the future work section names it precisely. Pipelining the unit is the single biggest step. Everything else follows from that.

8.7 Chapter Summary

This chapter collected the outcomes of the full project. Section 8.1 described what was built across the seven chapters. Section 8.2 identified the five concrete contributions. Section 8.3 presented the final cycle-count results at both kernel and model level. Section 8.4 stated the known limitations honestly, including the non-pipelined execution model, the absence of hardware interlocks, the missing area and power characterization. Section 8.5 described the natural next steps under four headings: Architecture, ISA, Software, and Evaluation. The chapter closed with a direct answer to the question the project started with: a targeted RVV co-processor on CV32E40P reduces inference time by 4.7–7.0× at the model level, with higher gains on compute-intensive kernels and lower gains where vector addressing overhead dominates the arithmetic cost.

Bibliography

[1] OpenHW Group. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture*. Document Version 20260417. 2026. URL: https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest.

[2] RISC-V International. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture*. Document Version 20240411. 2024. URL: https://riscv.org/specifications/.